Converting Linear-Time Temporal Logic to Generalized Büchi Automata

Alexander Schimpf and Peter Lammich

May 1, 2015

Abstract

We formalize linear-time temporal logic (LTL) and the algorithm by Gerth et al. to convert LTL formulas to generalized Büchi automata. We also formalize some syntactic rewrite rules that can be applied to optimize the LTL formula before conversion. Moreover, we integrate the Stuttering Equivalence AFP-Entry by Stefan Merz, adapting the lemma that next-free LTL formula cannot distinguish between stuttering equivalent runs to our setting.

We use the Isabelle Refinement and Collection framework, as well as the Autoref tool, to obtain a refined version of our algorithm, from which efficiently executable code can be extracted.

Contents

1 Introduction

In LTL model checking obtaining an equivalent automaton from a linear temporal logic (LTL) formula makes up an important nontrivial part of the whole process. Gerth et al. [2] present a simple tableau-based construction, which takes an LTL formula and decomposes it according to its structure gaining the desired automaton step-by-step.

In this entry, we formalize Linear Temporal Logic (LTL), some optimizing syntactic rewrite rules on LTL formulas, and Gerth's algorithm. Using the Isabelle Refinement Framework, we extract efficient code from our formalization.

Moreover, we connect our LTL formalization to the one of Stefan Merz [3], adapting the lemma that next-free LTL formula cannot distinguish between stuttering equivalent runs to our setting.

This work is part of the CAVA project [1] to implement an executable fully verified LTL model checker.

2 Linear Temporal Logic

theory *LTL*
imports
../CAVA-Automata/Words Refine-Util
begin

2.1 LTL formulas

2.1.1 Syntax

datatype
'a ltl = *LTLTrue*
| *LTLFalse*
| *LTLProp 'a*
| *LTLNeg 'a ltl*
| *LTLAnd 'a ltl 'a ltl*
| *LTLOr 'a ltl 'a ltl*
| *LTLNext 'a ltl*
| *LTLUntil 'a ltl 'a ltl*
| *LTLRelease 'a ltl 'a ltl*

The following locale defines syntactic sugar for parsing and printing LTL formulas in Isabelle

locale *LTL-Syntax* **begin**
notation
LTLTrue (*true*)
and *LTLFalse* (*false*)
and *LTLProp* (*prop*'(-'))

and *LTLNeg* (*not* - [*85*] *85*)
and *LTLAnd* (- *and* - [*82*,*82*] *81*)
and *LTLOr* (- *or* - [*81*,*81*] *80*)
and *LTLNext* (*X* - [*88*] *87*)
and *LTLUntil* (- *U* - [*84*,*84*] *83*)
and *LTLRelease* (- *V* - [*83*,*83*] *82*)
end

2.1.2 Semantics

We first provide an abstract semantics, that is parameterized with the semantics of atomic propositions

context begin interpretation *LTL-Syntax* **.**

primrec *ltl-semantics* :: *'ap set word* ⇒ *'ap ltl* ⇒ *bool*
(- ⊨ - [*80*,*80*] *80*)
where
ξ ⊨ *true* = *True*
| *ξ* ⊨ *false* = *False*
| *ξ* ⊨ *prop*(*q*) = (*q* ∈ (*ξ* *0*))
| *ξ* ⊨ *not* *φ* = (¬ *ξ* ⊨ *φ*)
| *ξ* ⊨ *φ* *and* *ψ* = (*ξ* ⊨ *φ* ∧ *ξ* ⊨ *ψ*)
| *ξ* ⊨ *φ* *or* *ψ* = (*ξ* ⊨ *φ* ∨ *ξ* ⊨ *ψ*)
| *ξ* ⊨ *X* *φ* = (*suffix* *1* *ξ* ⊨ *φ*)
| *ξ* ⊨ *φ* *U* *ψ* = (∃ *i*. *suffix* *i* *ξ* ⊨ *ψ* ∧ (∀ *j*<*i*. *suffix* *j* *ξ* ⊨ *φ*))
| *ξ* ⊨ *φ* *V* *ψ* = (∀ *i*. *suffix* *i* *ξ* ⊨ *ψ* ∨ (∃ *j*<*i*. *suffix* *j* *ξ* ⊨ *φ*))

definition *ltl-language* *φ* ≡ {*ξ*. *ξ* ⊨ *φ*}
end

2.1.3 Explicit Syntactic Sugar

In this section, we provide a formulation of LTL with explicit syntactic sugar deeply embedded. This formalization serves as a reference semantics.

datatype-new (*ltlc-aprops*: *'a*)
ltlc = *LTLcTrue*
| *LTLcFalse*
| *LTLcProp* *'a*
| *LTLcNeg* *'a ltlc*
| *LTLcAnd* *'a ltlc* *'a ltlc*
| *LTLcOr* *'a ltlc* *'a ltlc*
| *LTLcImplies* *'a ltlc* *'a ltlc*
| *LTLcIff* *'a ltlc* *'a ltlc*
| *LTLcNext* *'a ltlc*
| *LTLcFinal* *'a ltlc*
| *LTLcGlobal* *'a ltlc*
| *LTLcUntil* *'a ltlc* *'a ltlc*
| *LTLcRelease* *'a ltlc* *'a ltlc*

context *LTL-Syntax* **begin**
notation
LTLcTrue $(true_c)$
and *LTLcFalse* $(false_c)$
and *LTLcProp* $(prop_c\,'(-'))$
and *LTLcNeg* $(not_c$ - $[85]$ $85)$
and *LTLcAnd* $(-\ and_c$ - $[82,82]$ $81)$
and *LTLcOr* $(-\ or_c$ - $[81,81]$ $80)$
and *LTLcImplies* $(-\ implies_c$ - $[81,81]$ $80)$
and *LTLcIff* $(-\ iff_c$ - $[81,81]$ $80)$
and *LTLcNext* $(X_c$ - $[88]$ $87)$
and *LTLcFinal* $(F_c$ - $[88]$ $87)$
and *LTLcGlobal* $(G_c$ - $[88]$ $87)$
and *LTLcUntil* $(-\ U_c$ - $[84,84]$ $83)$
and *LTLcRelease* $(-\ V_c$ - $[83,83]$ $82)$
end

context begin interpretation *LTL-Syntax* **.**

primrec *ltlc-semantics*
:: $['a\ set\ word,\ 'a\ ltlc] \Rightarrow bool$ $(-\models_c$ - $[80,80]$ $80)$
where
$\xi \models_c true_c = True$
$|\ \xi \models_c false_c = False$
$|\ \xi \models_c prop_c(q) = (q{\in}\xi\ 0)$
$|\ \xi \models_c not_c\ \varphi = (\neg\ \xi \models_c \varphi)$
$|\ \xi \models_c \varphi\ and_c\ \psi = (\xi \models_c \varphi \wedge \xi \models_c \psi)$
$|\ \xi \models_c \varphi\ or_c\ \psi = (\xi \models_c \varphi \vee \xi \models_c \psi)$
$|\ \xi \models_c \varphi\ implies_c\ \psi = (\xi \models_c \varphi \longrightarrow \xi \models_c \psi)$
$|\ \xi \models_c \varphi\ iff_c\ \psi = (\xi \models_c \varphi \longleftrightarrow \xi \models_c \psi)$
$|\ \xi \models_c X_c\ \varphi = (suffix\ 1\ \xi \models_c \varphi)$
$|\ \xi \models_c F_c\ \varphi = (\exists\, i.\ suffix\ i\ \xi \models_c \varphi)$
$|\ \xi \models_c G_c\ \varphi = (\forall\, i.\ suffix\ i\ \xi \models_c \varphi)$
$|\ \xi \models_c \varphi\ U_c\ \psi = (\exists\, i.\ suffix\ i\ \xi \models_c \psi \wedge (\forall\, j{<}i.\ suffix\ j\ \xi \models_c \varphi))$
$|\ \xi \models_c \varphi\ V_c\ \psi = (\forall\, i.\ suffix\ i\ \xi \models_c \psi \vee (\exists\, j{<}i.\ suffix\ j\ \xi \models_c \varphi))$

definition *ltlc-language* $\varphi \equiv \{\xi.\ \xi \models_c \varphi\}$

lemma *ltlc-language-negate*[*simp*]:
ltlc-language $(not_c\ \varphi) = -$ *ltlc-language* φ
unfolding *ltlc-language-def*
by *auto*

lemma *ltlc-semantics-sugar*:
$\xi \models_c \varphi\ implies_c\ \psi = \xi \models_c (not_c\ \varphi\ or_c\ \psi)$
$\xi \models_c \varphi\ iff_c\ \psi = \xi \models_c ((not_c\ \varphi\ or_c\ \psi)\ and_c\ (not_c\ \psi\ or_c\ \varphi))$
$\xi \models_c F_c\ \varphi = \xi \models_c (true_c\ U_c\ \varphi)$
$\xi \models_c G_c\ \varphi = \xi \models_c (false_c\ V_c\ \varphi)$

by *auto*

definition *pw-eq-on S w w′* ≡ ∀ *i*. *w i* ∩ *S* = *w′ i* ∩ *S*

lemma
  pw-eq-on-refl[*simp*]: *pw-eq-on S w w*
 and *pw-eq-on-sym*: *pw-eq-on S w w′* ⟹ *pw-eq-on S w′ w*
 and *pw-eq-on-trans*[*trans*]:
  ⟦*pw-eq-on S w w′*; *pw-eq-on S w′ w″*⟧ ⟹ *pw-eq-on S w w″*
 unfolding *pw-eq-on-def* **by** *auto*

lemma *ltlc-eq-on*: *pw-eq-on* (*ltlc-aprops φ*) *w w′* ⟹ *w* ⊨*c φ* ⟷ *w′* ⊨*c φ*
 unfolding *pw-eq-on-def*
 apply (*induction φ arbitrary*: *w w′*)
 apply (*simp-all add*: *suffix-def*)
 apply (*auto*) []
 apply ((*rprems*, (*auto*) []) | *fo-rule arg-cong2 arg-cong* | *intro ext* | *simp*)+
 done

lemma *map-ltlc-semantics-aux*:
 assumes *inj-on f APs*
 assumes ⋃(*range ξ*) ⊆ *APs*
 assumes *ltlc-aprops φ* ⊆ *APs*
 shows *ξ* ⊨*c φ* ⟷ (λ*i*. *f* ‘ *ξ i*) ⊨*c map-ltlc f φ*
 using *assms*(*2*,*3*)
 apply (*induct φ arbitrary*: *ξ*)
 using *assms*(*1*)
 apply (*simp-all add*: *suffix-def inj-on-Un*)
 apply (*auto dest*: *inj-onD*) []
 apply ((*rprems*, (*auto*) []) | *fo-rule arg-cong2 arg-cong* | *intro ext* | *simp*)+
 done

definition *map-aprops f APs* ≡ { *i*. ∃ *p*∈*APs*. *f p* = *Some i* }

lemma *map-ltlc-semantics*:
 assumes *INJ*: *inj-on f* (*dom f*) **and** *DOM*: *ltlc-aprops φ* ⊆ *dom f*
 shows *ξ* ⊨*c φ* ⟷ (*map-aprops f o ξ*) ⊨*c map-ltlc* (*the o f*) *φ*
proof −
 let *?ξr* = λ*i*. *ξ i* ∩ *ltlc-aprops φ*
 let *?ξr′* = λ*i*. *ξ i* ∩ *dom f*

 have *1*: ⋃*range ?ξr* ⊆ *ltlc-aprops φ* **by** *auto*

 have *INJ-the-dom*: *inj-on* (*the o f*) (*dom f*)
  using *assms*
  by (*auto simp*: *inj-on-def domIff*)
 note *2* = *subset-inj-on*[*OF this DOM*]

```
  have 3: (λi. (the o f) ‘ ?ξr' i) = map-aprops f o ξ using DOM INJ
    apply (auto intro!: ext simp: map-aprops-def domIff image-iff)
    by (metis Int-iff domI option.sel)

  have ξ ⊨c φ ⟷ ?ξr ⊨c φ
    apply (rule ltlc-eq-on)
    apply (auto simp: pw-eq-on-def)
    done
  also from map-ltlc-semantics-aux[OF 2 1 subset-refl]
  have ... ⟷ (λi. (the o f) ‘ ?ξr i) ⊨c map-ltlc (the o f) φ .
  also have ... ⟷ (λi. (the o f) ‘ ?ξr' i) ⊨c map-ltlc (the o f) φ
    apply (rule ltlc-eq-on) using DOM INJ
    apply (auto simp: pw-eq-on-def ltlc.set-map domIff image-iff)
    by (metis Int-iff contra-subsetD domD domI inj-on-eq-iff option.sel)
  also note 3
  finally show ?thesis .
qed

lemma map-ltlc-semantics-inv:
  assumes INJ: inj-on f (dom f) and DOM: ltlc-aprops φ ⊆ dom f
  shows ξ ⊨c map-ltlc (the o f) φ ⟷ (λi. (the o f) −‘ ξ i) ⊨c φ
  using map-ltlc-semantics[OF assms]
  apply simp
  apply (intro ltlc-eq-on)
  apply (auto simp add: pw-eq-on-def ltlc.set-map map-aprops-def)
  by (metis DOM comp-apply contra-subsetD domD option.sel vimage-eq)
```

Conversion from LTL with common syntax to LTL

```
fun ltlc-to-ltl :: 'a ltlc ⇒ 'a ltl
where
  ltlc-to-ltl true_c = true
| ltlc-to-ltl false_c = false
| ltlc-to-ltl prop_c(q) = prop(q)
| ltlc-to-ltl (not_c φ) = not (ltlc-to-ltl φ)
| ltlc-to-ltl (φ and_c ψ) = ltlc-to-ltl φ and ltlc-to-ltl ψ
| ltlc-to-ltl (φ or_c ψ) = ltlc-to-ltl φ or ltlc-to-ltl ψ
| ltlc-to-ltl (φ implies_c ψ) = (not (ltlc-to-ltl φ)) or (ltlc-to-ltl ψ)
| ltlc-to-ltl (φ iff_c ψ) = (let φ' = ltlc-to-ltl φ in
                              let ψ' = ltlc-to-ltl ψ in
                              (not φ' or ψ') and (not ψ' or φ'))
| ltlc-to-ltl (X_c φ) = X (ltlc-to-ltl φ)
| ltlc-to-ltl (F_c φ) = true U ltlc-to-ltl φ
| ltlc-to-ltl (G_c φ) = false V ltlc-to-ltl φ
| ltlc-to-ltl (φ U_c ψ) = ltlc-to-ltl φ U ltlc-to-ltl ψ
| ltlc-to-ltl (φ V_c ψ) = ltlc-to-ltl φ V ltlc-to-ltl ψ

lemma ltlc-to-ltl-equiv:
  ξ ⊨ (ltlc-to-ltl φ) ⟷ ξ ⊨c φ
  apply (induct φ arbitrary:ξ)
```

apply (*auto simp*: *Let-def*)
done

end

2.2 Semantic Preserving Syntax Transformations

context begin interpretation *LTL-Syntax* **.**

lemma *ltl-true-or-con*[*simp*]:
$\xi \models prop(p)\ or\ (not\ prop(p)) \longleftrightarrow \xi \models true$
by *auto*

lemma *ltl-false-true-con*[*simp*]:
$\xi \models not\ true \longleftrightarrow \xi \models false$
by *auto*

The negation symbol can be passed through the next operator.

lemma *ltl-Next-Neg-con*[*simp*]:
$\xi \models X\ (not\ \varphi) \longleftrightarrow \xi \models not\ X\ \varphi$
by *auto*

The connection between Until and Release

lemma *ltl-Release-Until-con*:
$\xi \models \varphi\ V\ \psi \longleftrightarrow (\neg\ \xi \models (not\ \varphi)\ U\ (not\ \psi))$
by *auto*

Expand strategy

lemma *ltl-expand-Until*:
$\xi \models \varphi\ U\ \psi \longleftrightarrow (\xi \models \psi\ or\ (\varphi\ and\ (X\ (\varphi\ U\ \psi))))$ (**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
then obtain *i*
where *psi-is*: *suffix i* $\xi \models \psi$
and *phi-is*: $\forall j<i.$ *suffix j* $\xi \models \varphi$ **by** *auto*
show *?rhs*
proof(*cases i*)
case *0*
thus *?rhs* **using** *psi-is* **by** *auto*
next
case (*Suc k*)
with *phi-is* **have** $\xi \models \varphi$ **by** *auto*
moreover
have $\xi \models X\ (\varphi\ U\ \psi)$
using *psi-is phi-is Suc* **by** *auto*
ultimately show *?rhs* **by** *auto*
qed
next
assume *rhs*: *?rhs*

show *?lhs*
proof(*cases* $\xi \models \psi$)
case *True*
hence *suffix 0* $\xi \models \psi$ **by** *auto*
moreover
have $\forall j{<}0.$ *suffix* j $\xi \models \varphi$ **by** *auto*
ultimately
have $\exists i.$ *suffix* i $\xi \models \psi$
$\wedge$ $(\forall j{<}i.$ *suffix* j $\xi \models \varphi)$ **by** *blast*
thus *?lhs* **by** *auto*
next
case *False*
hence *phi-is*: $\xi \models \varphi$
and $\xi \models X$ $(\varphi$ U $\psi)$ **using** *rhs* **by** *auto*
then obtain i
where *psi-suc-is*: *suffix* $(Suc\ i)$ $\xi \models \psi$
and *phi-suc-is*: $\forall j{<}i.$ *suffix* $(Suc\ j)$ $\xi \models \varphi$ **by** *auto*
have *sbgoal*: $\forall j{<}(Suc\ i).$ *suffix* j $\xi \models \varphi$
proof(*clarify*)
fix j
assume *j-less*: $j{<}Suc\ i$
show *suffix* j $\xi \models \varphi$
proof (*cases* j)
assume $j{=}0$
thus *?thesis* **using** *phi-is* **by** *auto*
next
fix k
assume $j{=}Suc\ k$
thus *?thesis* **using** *j-less phi-suc-is* **by** *auto*
qed
qed
thus *?lhs* **using** *psi-suc-is phi-is* **by** *auto*
qed
qed

lemma *ltl-expand-Release*:
$\xi \models \varphi$ V $\psi \longleftrightarrow$ $(\xi \models \psi$ *and* $(\varphi$ *or* $(X$ $(\varphi$ V $\psi))))$
proof −
from *ltl-expand-Until*[*of* ξ *not* φ *not* ψ]
show *?thesis* **by** *auto*
qed

Double negation structure of an LTL formula

lemma [*simp*]:
not $((\lambda\mu.$ *not not* $\mu)$ ^^ $n)$ $\varphi = ((\lambda\mu.$ *not not* $\mu)$ ^^ $n)$ $(not\ \varphi)$
by (*induct* n) *auto*

lemma *ltl-double-neg-struct*:
shows $\exists n\ \psi.\ \varphi = ((\lambda\xi.$ *not not* $\xi)$ ^^ $n)$ $\psi \wedge (\forall \nu.\ \psi \neq not\ not\ \nu)$

```
  (is ∃ n ψ. ?Q φ n ψ)
proof(cases ∀ ν. φ ≠ not ν)
  case goal1
    hence ?Q φ 0 φ by auto
    thus ?thesis by blast
next
  case goal2
    thus ?thesis
    proof(induct φ)
      case (LTLNeg φ')
        thus ?case
        proof(cases ∀ ν. φ' ≠ not ν)
          case goal1
            hence ?Q (not φ') 0 (not φ') by auto
            thus ?case by blast
        next
          case goal2
            then obtain n' ψ' where ?Q φ' n' ψ' by auto
            thus ?case
            proof(cases ∃ ψ''. ψ' = not ψ'')
              case goal1
                then obtain ψ''
                      where ψ' = not ψ'' by auto
                hence ?Q (not φ') (n'+1) ψ'' using goal1 by auto
                thus ?case by blast
            next
              case goal2
                hence ?Q (not φ') n' (not ψ') by auto
                thus ?case by blast
            qed
        qed
    qed auto
qed

lemma ltl-size-double-neg:
  assumes ψ = ((λμ. not not μ) ^^ n) φ
    shows size φ ≤ size ψ
using assms proof (induct n arbitrary:φ ψ)
  case (Suc k)
    obtain μ where μ-eq: μ = ((λμ. not not μ) ^^ k) φ by auto
    hence ψ = not not μ using Suc by auto
    moreover have size φ ≤ size μ using Suc μ-eq by auto
    ultimately show ?case by auto
qed auto
```

Pushing negation to the top of a proposition

```
fun
  ltl-pushneg :: 'a ltl ⇒ 'a ltl
  where
```

```
  ltl-pushneg true = true
| ltl-pushneg false = false
| ltl-pushneg prop(q) = prop(q)
| ltl-pushneg (not true) = false
| ltl-pushneg (not false) = true
| ltl-pushneg (not prop(q)) = not prop(q)
| ltl-pushneg (not (not ψ)) = ltl-pushneg ψ
| ltl-pushneg (not (ν and μ)) = ltl-pushneg (not ν) or ltl-pushneg (not μ)
| ltl-pushneg (not (ν or μ)) = ltl-pushneg (not ν) and ltl-pushneg (not μ)
| ltl-pushneg (not (X ψ)) = X ltl-pushneg (not ψ)
| ltl-pushneg (not (ν U μ)) = ltl-pushneg (not ν) V ltl-pushneg (not μ)
| ltl-pushneg (not (ν V μ)) = ltl-pushneg (not ν) U ltl-pushneg (not μ)
| ltl-pushneg (φ and ψ) = (ltl-pushneg φ) and (ltl-pushneg ψ)
| ltl-pushneg (φ or ψ) = (ltl-pushneg φ) or (ltl-pushneg ψ)
| ltl-pushneg (X φ) = X (ltl-pushneg φ)
| ltl-pushneg (φ U ψ) = (ltl-pushneg φ) U (ltl-pushneg ψ)
| ltl-pushneg (φ V ψ) = (ltl-pushneg φ) V (ltl-pushneg ψ)
```

In fact, the *ltl-pushneg* function does not change the semantics of the input formula.

```
lemma ltl-pushneg-neg:
  shows ξ ⊨ ltl-pushneg (not φ) ⟷ ξ ⊨ not ltl-pushneg φ
  by (induct φ arbitrary: ξ) auto

theorem ltl-pushneg-equiv[simp]:
  ξ ⊨ ltl-pushneg φ ⟷ ξ ⊨ φ
proof (induct φ arbitrary: ξ)
  case (LTLNeg ψ)
  with ltl-pushneg-neg show ?case by auto
qed auto
```

We can now show that *ltl-pushneg* does what it should do. Actually the negation occurs after the transformation only on top of a proposition.

```
lemma ltl-pushneg-double-neg:
  shows ltl-pushneg (((λφ. not not φ) ^^ n) φ) = ltl-pushneg φ
by (induct n arbitrary:φ) auto

lemma ltl-pushneg-neg-struct:
  assumes ltl-pushneg φ = not ψ
    shows ∃ q. ψ = prop(q)
proof −
  from ltl-double-neg-struct
  obtain n μ where φ-eq: φ = ((λμ. not not μ) ^^ n) μ
               and μ-neq: (∀ ν. μ ≠ not not ν) by blast
  with ltl-pushneg-double-neg have ltl-pushneg φ = ltl-pushneg μ by auto
  thus ?thesis using φ-eq assms μ-neq
  proof(induct μ)
    case (LTLNeg f) thus ?case by (cases f) auto
  qed auto
```

qed

inductive *subfrml*
where
  subfrml φ (*not* φ)
| *subfrml* φ (φ *and* ψ)
| *subfrml* ψ (φ *and* ψ)
| *subfrml* φ (φ *or* ψ)
| *subfrml* ψ (φ *or* ψ)
| *subfrml* φ (*X* φ)
| *subfrml* φ (φ *U* ψ)
| *subfrml* ψ (φ *U* ψ)
| *subfrml* φ (φ *V* ψ)
| *subfrml* ψ (φ *V* ψ)

abbreviation *is-subfrml* (- *is'-subformula'-of* -)
where
  is-subfrml ψ $\varphi \equiv$ *subfrml*** ψ φ

lemma *subfrml-size*:
  assumes *subfrml* ψ φ
    shows *size* $\psi <$ *size* φ
using *assms* **by** (*induct* φ) *auto*

lemma *subformula-size*:
  assumes ψ *is-subformula-of* φ
    shows *size* $\psi <$ *size* $\varphi \vee \psi = \varphi$
using *assms* **proof**(*induct* φ)
  case *base* **thus** *?case* **by** *auto*
next
  case (*step* ν μ)
    hence *size* $\nu <$ *size* μ **by** (*rule-tac subfrml-size*)
    thus *?case* **using** *step* **by** *auto*
qed

lemma *subformula-on-ltl-pushneg*:
  assumes ψ *is-subformula-of* (*ltl-pushneg* φ)
    shows $\exists \mu.\ \psi =$ *ltl-pushneg* μ
proof(*cases* $\psi =$ *ltl-pushneg* φ)
  case *True* **thus** *?thesis* **by** *blast*
next
  case *False* **thus** *?thesis* **using** *assms*
  proof(*induct* φ *rule*:*ltl-pushneg.induct*)
    case *goal1* **thus** *?case* **using** *subformula-size* **by** *force*
  next
    case *goal2* **thus** *?case* **using** *subformula-size* **by** *force*
  next
    case *goal3* **thus** *?case* **using** *subformula-size* **by** *force*

```
next
  case goal4 thus ?case using subformula-size by force
next
  case goal5 thus ?case using subformula-size by force
next
  case (goal6 q)
    let ?frml = not prop(q)
    from rtrancl-eq-or-trancl[to-pred, of subfrml]
    have t-prm: subfrml^++ ψ ?frml
    using goal6 by auto
    obtain μ
     where sf-prm: subfrml ψ  μ
       and rt-prm: μ is-subformula-of ?frml
    using tranclpD[OF t-prm] by blast
    show ?case
    proof(cases μ = ?frml)
      assume μ = ?frml
      with sf-prm have ψ = ltl-pushneg prop(q)
      by (cases ψ) auto
      thus ?thesis by blast
    next
      assume μ ≠ ?frml
      with rtranclpD[OF rt-prm]
           tranclp-into-rtranclp[of subfrml]
           subformula-size[of μ ?frml]
      have size μ = 0 by auto
      with subfrml-size[OF sf-prm]
      show ?thesis by auto
    qed
next
  case goal7 thus ?case by auto
next
  case (goal8 ν μ)
    let ?frml = not (ν and μ)
    from tranclD2[to-pred, of subfrml ψ ltl-pushneg ?frml]
         rtrancl-eq-or-trancl[to-pred, of subfrml]
    obtain z
     where subfrml z (ltl-pushneg ?frml)
       and rt-prm: ψ is-subformula-of z
     using goal8 by auto
    hence z-is:
          z = ltl-pushneg (not ν) ∨
           z = ltl-pushneg (not μ) by (cases z) auto
    show ?case
    proof(cases ψ = z)
      assume ψ = z
      with z-is show ?thesis by auto
    next
      assume ψ ≠ z
```

```
      with rtranclpD[OF rt-prm]
          tranclp-into-rtranclp
      have ψ is-subformula-of z by auto
      thus ?thesis using goal8 z-is by auto
    qed
next
  case (goal9 ν μ)
    let ?frml = not (ν or μ)
    from tranclD2[to-pred, of subfrml ψ ltl-pushneg ?frml]
        rtrancl-eq-or-trancl[to-pred, of subfrml]
    obtain z
     where subfrml z (ltl-pushneg ?frml)
       and rt-prm: ψ is-subformula-of z
     using goal9 by auto
    hence z-is:
         z = ltl-pushneg (not ν) ∨
          z = ltl-pushneg (not μ) by (cases z) auto
    show ?case
    proof(cases ψ = z)
      assume ψ = z
      with z-is show ?thesis by auto
    next
      assume ψ ≠ z
      with rtranclpD[OF rt-prm]
          tranclp-into-rtranclp
      have ψ is-subformula-of z by auto
      thus ?thesis using goal9 z-is by auto
    qed
next
  case (goal10 μ)
    let ?frml = not (X μ)
    from tranclD2[to-pred, of subfrml ψ ltl-pushneg ?frml]
        rtrancl-eq-or-trancl[to-pred, of subfrml]
    obtain z
     where subfrml z (ltl-pushneg ?frml)
       and rt-prm: ψ is-subformula-of z
     using goal10 by auto
    hence z-is: z = ltl-pushneg (not μ)
    by (cases z) auto
    show ?case
    proof(cases ψ = z)
      assume ψ = z
      with z-is show ?thesis by auto
    next
      assume ψ ≠ z
      with rtranclpD[OF rt-prm]
          tranclp-into-rtranclp
      have ψ is-subformula-of z by auto
      thus ?thesis using goal10 z-is by auto
```

```
    qed
next
  case (goal11 ν μ)
    let ?frml = not (ν U μ)
    from tranclD2[to-pred, of subfrml ψ ltl-pushneg ?frml]
         rtrancl-eq-or-trancl[to-pred, of subfrml]
    obtain z
     where subfrml z (ltl-pushneg ?frml)
       and rt-prm: ψ is-subformula-of z
     using goal11 by auto
    hence z-is:
          z = ltl-pushneg (not ν) ∨
           z = ltl-pushneg (not μ) by (cases z) auto
    show ?case
    proof(cases ψ = z)
      assume ψ = z
      with z-is show ?thesis by auto
    next
      assume ψ ≠ z
      with rtranclpD[OF rt-prm]
           tranclp-into-rtranclp
      have ψ is-subformula-of z by auto
      thus ?thesis using goal11 z-is by auto
    qed
next
  case (goal12 ν μ)
    let ?frml = not (ν V μ)
    from tranclD2[to-pred, of subfrml ψ ltl-pushneg ?frml]
         rtrancl-eq-or-trancl[to-pred, of subfrml]
    obtain z
     where subfrml z (ltl-pushneg ?frml)
       and rt-prm: ψ is-subformula-of z
     using goal12 by auto
    hence z-is:
          z = ltl-pushneg (not ν) ∨
           z = ltl-pushneg (not μ) by (cases z) auto
    show ?case
    proof(cases ψ = z)
      assume ψ = z
      with z-is show ?thesis by auto
    next
      assume ψ ≠ z
      with rtranclpD[OF rt-prm]
           tranclp-into-rtranclp
      have ψ is-subformula-of z by auto
      thus ?thesis using goal12 z-is by auto
    qed
next
  case (goal13 ν μ)
```

```
    let ?frml = ν and μ
    from tranclD2[to-pred, of subfrml ψ ltl-pushneg ?frml]
        rtrancl-eq-or-trancl[to-pred, of subfrml]
    obtain z
     where subfrml z (ltl-pushneg ?frml)
       and rt-prm: ψ is-subformula-of z
     using goal13 by auto
    hence z-is:
          z = ltl-pushneg ν ∨
           z = ltl-pushneg μ by (cases z) auto
    show ?case
    proof(cases ψ = z)
      assume ψ = z
      with z-is show ?thesis by auto
    next
      assume ψ ≠ z
      with rtranclpD[OF rt-prm]
           tranclp-into-rtranclp
      have ψ is-subformula-of z by auto
      thus ?thesis using goal13 z-is by auto
    qed
next
  case (goal14 ν μ)
    let ?frml = ν or μ
    from tranclD2[to-pred, of subfrml ψ ltl-pushneg ?frml]
        rtrancl-eq-or-trancl[to-pred, of subfrml]
    obtain z
     where subfrml z (ltl-pushneg ?frml)
       and rt-prm: ψ is-subformula-of z
     using goal14 by auto
    hence z-is:
          z = ltl-pushneg ν ∨
           z = ltl-pushneg μ by (cases z) auto
    show ?case
    proof(cases ψ = z)
      assume ψ = z
      with z-is show ?thesis by auto
    next
      assume ψ ≠ z
      with rtranclpD[OF rt-prm]
           tranclp-into-rtranclp
      have ψ is-subformula-of z by auto
      thus ?thesis using goal14 z-is by auto
    qed
next
  case (goal15 μ)
    let ?frml = X μ
    from tranclD2[to-pred, of subfrml ψ ltl-pushneg ?frml]
        rtrancl-eq-or-trancl[to-pred, of subfrml]
```

```
    obtain z
     where subfrml z (ltl-pushneg ?frml)
       and rt-prm: ψ is-subformula-of z
     using goal15 by auto
    hence z-is: z = ltl-pushneg μ by (cases z) auto
    show ?case
    proof(cases ψ = z)
      assume ψ = z
      with z-is show ?thesis by auto
    next
      assume ψ ≠ z
      with rtranclpD[OF rt-prm]
           tranclp-into-rtranclp
      have ψ is-subformula-of z by auto
      thus ?thesis using goal15 z-is by auto
    qed
next
  case (goal16 ν μ)
    let ?frml = ν U μ
    from tranclD2[to-pred, of subfrml ψ ltl-pushneg ?frml]
         rtrancl-eq-or-trancl[to-pred, of subfrml]
    obtain z
     where subfrml z (ltl-pushneg ?frml)
       and rt-prm: ψ is-subformula-of z
     using goal16 by auto
    hence z-is:
          z = ltl-pushneg ν ∨
           z = ltl-pushneg μ by (cases z) auto
    show ?case
    proof(cases ψ = z)
      assume ψ = z
      with z-is show ?thesis by auto
    next
      assume ψ ≠ z
      with rtranclpD[OF rt-prm]
           tranclp-into-rtranclp
      have ψ is-subformula-of z by auto
      thus ?thesis using goal16 z-is by auto
    qed
next
  case (goal17 ν μ)
    let ?frml = ν V μ
    from tranclD2[to-pred, of subfrml ψ ltl-pushneg ?frml]
         rtrancl-eq-or-trancl[to-pred, of subfrml]
    obtain z
     where subfrml z (ltl-pushneg ?frml)
       and rt-prm: ψ is-subformula-of z
     using goal17 by auto
    hence z-is:
```

```
        z = ltl-pushneg ν ∨
         z = ltl-pushneg μ by (cases z) auto
    show ?case
    proof(cases ψ = z)
      assume ψ = z
      with z-is show ?thesis by auto
    next
      assume ψ ≠ z
      with rtranclpD[OF rt-prm]
          tranclp-into-rtranclp
      have ψ is-subformula-of z by auto
      thus ?thesis using goal17 z-is by auto
    qed
  qed
qed
```

The fact that after pushing the negation the structure of a formula changes, is shown by the following theorem. Indeed, after pushing the negation symbol inside a formula, it occurs at most on top of a proposition.

```
theorem ltl-pushneg-struct:
  assumes (not ψ) is-subformula-of (ltl-pushneg φ)
    shows ∃ q. ψ = prop(q)
proof −
  from assms subformula-on-ltl-pushneg
  obtain μ
   where prm:not ψ = ltl-pushneg μ by blast
  from ltl-pushneg-neg-struct[OF sym[OF prm]]
  show ?thesis by auto
qed
```

Now we want to show that the size of the formula, which is transformed by *ltl-pushneg*, does not increase 'too much', i.e. there is no exponential blowup produced by the transformation. For that purpose we need an additional function, which counts the literals of the derivation tree of a formula. The idea is, that, assuming the worst case, the pushing of negation can only increase the size of a formula by putting the negation symbol on top of every proposition inside the formula.

```
fun leafcnt :: 'a ltl ⇒ nat
where
  leafcnt true = 1
| leafcnt false = 1
| leafcnt prop(q) = 1
| leafcnt (not φ) = leafcnt φ
| leafcnt (φ and ψ) = (leafcnt φ) + (leafcnt ψ)
| leafcnt (φ or ψ) = (leafcnt φ) + (leafcnt ψ)
| leafcnt (X φ) = leafcnt φ
| leafcnt (φ U ψ) = (leafcnt φ) + (leafcnt ψ)
| leafcnt (φ V ψ) = (leafcnt φ) + (leafcnt ψ)
```

lemma *leafcnt-double-neg-ident*:
leafcnt (((λμ. *not not* μ) ^^ *n*) φ) = *leafcnt* φ
by (*induct n arbitrary*:φ) *auto*

lemma *ltl-pushneg-help*:
∃φ. *ltl-pushneg* ψ = *ltl-pushneg* φ
∧ ((∃ν. φ = *not* ν ∧ (∀μ. ν ≠ *not* μ)) ∨ (∀μ. φ ≠ *not* μ))
∧ *size* φ ≤ *size* ψ
∧ *leafcnt* φ = *leafcnt* ψ
(**is** ∃φ. *?P* ψ φ ∧ *?Q* φ ∧ *?R* φ)
proof −
from *ltl-double-neg-struct*
obtain *n* φ **where** *dblneg*: ψ = ((λμ. *not not* μ) ^^ *n*) φ
and φ-*neq*: (∀ν. φ ≠ *not not* ν)
by *blast*
have *?Q* φ
proof(*rule ccontr*)
assume ¬ *?Q* φ
thus *False*
proof(*cases* ∃μ. φ = *not* μ)
case *goal1*
then obtain μ **where** φ = *not* μ **by** *blast*
with *goal1 dblneg* φ-*neq* **show** *?case* **by** *auto*
qed *auto*
qed
with *dblneg* φ-*neq*
ltl-pushneg-double-neg[*of n* φ]
leafcnt-double-neg-ident[*of n* φ]
ltl-size-double-neg[*OF dblneg*] **show** *?thesis* **by** *auto*
qed

lemma *ltl-pushneg-size-lin-help*:
assumes ψ = *ltl-pushneg* φ
shows *size* ψ + *1* ≤ *size* φ + *leafcnt* φ
using *assms* **proof** (*induct* ψ *arbitrary*: φ)
case *goal1* **show** *?case* **by** (*cases* φ) *auto*
next
case *goal2* **show** *?case* **by** (*cases* φ) *auto*
next
case *goal3* **show** *?case* **by** (*cases* φ) *auto*
next
case (*goal4* ψ′)
with *ltl-pushneg-neg-struct*[*of* φ ψ′] **obtain** *q* **where** ψ′ = *prop*(*q*) **by** *auto*
moreover
with *ltl-pushneg-help*[*of* φ]
obtain φ′ **where** *ltl-pushneg* φ = *ltl-pushneg* φ′

```
            and (∃ν. φ' = not ν ∧ (∀μ. ν ≠ not μ)) ∨ (∀μ. φ' ≠ not μ)
            and size φ' ≤ size φ
            and leafcnt φ' = leafcnt φ by auto
    ultimately show ?case using goal4
    proof(cases ∃ν. φ' = not ν ∧ (∀μ. ν ≠ not μ))
      case goal1
        then obtain ν where φ'is:φ' = not ν and ∀μ. ν ≠ not μ by auto
        thus ?case using goal1 by (cases ν) auto
    next
      case goal2 thus ?case by (cases φ') force+
    qed
next
  case (goal5 f g)
    with ltl-pushneg-help[of φ]
    obtain φ' where ltl-pushneg φ = ltl-pushneg φ'
            and (∃ν. φ' = not ν ∧ (∀μ. ν ≠ not μ)) ∨ (∀μ. φ' ≠ not μ)
            and size φ' ≤ size φ
            and leafcnt φ' = leafcnt φ by auto
    with goal5 show ?case
    proof(cases ∃ν. φ' = not ν ∧ (∀μ. ν ≠ not μ))
      case goal1
        then obtain ν where φ'is:φ' = not ν and ∀μ. ν ≠ not μ by auto
        hence size (f and g) ≤ size ν + leafcnt ν using goal1 by (cases ν) force+
        thus ?case using goal1 φ'is by auto
    next
      case goal2 thus ?case by (cases φ') force+
    qed
next
  case (goal6 f g)
    with ltl-pushneg-help[of φ]
    obtain φ' where ltl-pushneg φ = ltl-pushneg φ'
            and (∃ν. φ' = not ν ∧ (∀μ. ν ≠ not μ)) ∨ (∀μ. φ' ≠ not μ)
            and size φ' ≤ size φ
            and leafcnt φ' = leafcnt φ by auto
    with goal6 show ?case
    proof(cases ∃ν. φ' = not ν ∧ (∀μ. ν ≠ not μ))
      case goal1
        then obtain ν where φ'is:φ' = not ν and ∀μ. ν ≠ not μ by auto
        hence size (f or g) ≤ size ν + leafcnt ν using goal1 by (cases ν) force+
        thus ?case using goal1 φ'is by auto
    next
      case goal2 thus ?case by (cases φ') force+
    qed
next
  case (goal7 f)
    with ltl-pushneg-help[of φ]
    obtain φ' where ltl-pushneg φ = ltl-pushneg φ'
            and (∃ν. φ' = not ν ∧ (∀μ. ν ≠ not μ)) ∨ (∀μ. φ' ≠ not μ)
            and size φ' ≤ size φ
```

```
            and leafcnt φ' = leafcnt φ by auto
    with goal7 show ?case
    proof(cases ∃ν. φ' = not ν ∧ (∀μ. ν ≠ not μ))
      case goal1
        then obtain ν where φ'is:φ' = not ν and ∀μ. ν ≠ not μ by auto
        with goal1 show ?case by (cases ν) force+
    next
      case goal2 thus ?case by (cases φ') force+
    qed
next
  case (goal8 f g)
    with ltl-pushneg-help[of φ]
    obtain φ' where ltl-pushneg φ = ltl-pushneg φ'
            and (∃ν. φ' = not ν ∧ (∀μ. ν ≠ not μ)) ∨ (∀μ. φ' ≠ not μ)
            and size φ' ≤ size φ
            and leafcnt φ' = leafcnt φ by auto
    with goal8 show ?case
    proof(cases ∃ν. φ' = not ν ∧ (∀μ. ν ≠ not μ))
      case goal1
        then obtain ν where φ'is:φ' = not ν and ∀μ. ν ≠ not μ by auto
        hence size (f U g) ≤ size ν + leafcnt ν using goal1 by (cases ν) force+
        thus ?case using goal1 φ'is by auto
    next
      case goal2 thus ?case by (cases φ') force+
    qed
next
  case (goal9 f g)
    with ltl-pushneg-help[of φ]
    obtain φ' where ltl-pushneg φ = ltl-pushneg φ'
            and (∃ν. φ' = not ν ∧ (∀μ. ν ≠ not μ)) ∨ (∀μ. φ' ≠ not μ)
            and size φ' ≤ size φ
            and leafcnt φ' = leafcnt φ by auto
    with goal9 show ?case
    proof(cases ∃ν. φ' = not ν ∧ (∀μ. ν ≠ not μ))
      case goal1
        then obtain ν where φ'is:φ' = not ν and ∀μ. ν ≠ not μ by auto
        hence size (f V g) ≤ size ν + leafcnt ν using goal1 by (cases ν) force+
        thus ?case using goal1 φ'is by auto
    next
      case goal2 thus ?case by (cases φ') force+
    qed
qed

theorem ltl-pushneg-size-lin:
  size (ltl-pushneg φ) ≤ 2 * size φ
proof −
  have leafcnt φ ≤ size φ + 1 by (induct φ) auto
  with ltl-pushneg-size-lin-help[of - φ]
  have size (ltl-pushneg φ) + 1 ≤ size φ + size φ + 1 by force
```

thus *?thesis* **by** *auto*
qed

end

2.3 LTL formula in negation normal form (NNF)

We define a type of LTL formula in negation normal form (NNF)

datatype
'a ltln = *LTLnTrue*
| *LTLnFalse*
| *LTLnProp 'a*
| *LTLnNProp 'a*
| *LTLnAnd 'a ltln 'a ltln*
| *LTLnOr 'a ltln 'a ltln*
| *LTLnNext 'a ltln*
| *LTLnUntil 'a ltln 'a ltln*
| *LTLnRelease 'a ltln 'a ltln*

context *LTL-Syntax* **begin**
notation
LTLnTrue ($true_n$)
and *LTLnFalse* ($false_n$)
and *LTLnProp* ($prop_n$'(-'))
and *LTLnNProp* ($nprop_n$'(-'))
and *LTLnAnd* (- and_n - [*82*,*82*] *81*)
and *LTLnOr* (- or_n - [*84*,*84*] *83*)
and *LTLnNext* (X_n - [*88*] *87*)
and *LTLnUntil* (- U_n - [*84*,*84*] *83*)
and *LTLnRelease* (- V_n - [*84*,*84*] *83*)

abbreviation *ltln-eventuality* :: *'a ltln* ⇒ *'a ltln* ($\Diamond_n$ - [*88*] *87*)
where *ltln-eventuality* $\varphi \equiv true_n\ U_n\ \varphi$

abbreviation *ltln-universality* :: *'a ltln* ⇒ *'a ltln* ($\Box_n$ - [*88*] *87*)
where *ltln-universality* $\varphi \equiv false_n\ V_n\ \varphi$

end

context begin interpretation *LTL-Syntax* **.**

primrec *ltln-semantics* :: [*'a set word*, *'a ltln*] ⇒ *bool*
(- $\models_n$ - [*80*,*80*] *80*)
where
$\xi \models_n true_n = True$
| $\xi \models_n false_n = False$
| $\xi \models_n prop_n(q) = (q \in \xi\ 0)$
| $\xi \models_n nprop_n(q) = (q \notin \xi\ 0)$
| $\xi \models_n \varphi\ and_n\ \psi = (\xi \models_n \varphi \wedge \xi \models_n \psi)$

$| \ \xi \models_n \varphi \ or_n \ \psi = (\xi \models_n \varphi \vee \xi \models_n \psi)$
$| \ \xi \models_n X_n \ \varphi = (\mathit{suffix}\ 1\ \xi \models_n \varphi)$
$| \ \xi \models_n \varphi \ U_n \ \psi = (\exists\, i.\ \mathit{suffix}\ i\ \xi \models_n \psi \wedge (\forall\, j{<}i.\ \mathit{suffix}\ j\ \xi \models_n \varphi))$
$| \ \xi \models_n \varphi \ V_n \ \psi = (\forall\, i.\ \mathit{suffix}\ i\ \xi \models_n \psi \vee (\exists\, j{<}i.\ \mathit{suffix}\ j\ \xi \models_n \varphi))$

definition *ltln-language* $\varphi \equiv \{\xi.\ \xi \models_n \varphi\}$

Conversion from LTL to LTL in NNF

fun *ltl-to-ltln* :: *'a ltl* ⇒ *'a ltln*
where
ltl-to-ltln true = $true_n$
| *ltl-to-ltln false* = $false_n$
| *ltl-to-ltln prop*(q) = $prop_n(q)$
| *ltl-to-ltln* (*not prop*(q)) = $nprop_n(q)$
| *ltl-to-ltln* (φ *and* ψ) = *ltl-to-ltln* φ and_n *ltl-to-ltln* ψ
| *ltl-to-ltln* (φ *or* ψ) = *ltl-to-ltln* φ or_n *ltl-to-ltln* ψ
| *ltl-to-ltln* (X φ) = X_n (*ltl-to-ltln* φ)
| *ltl-to-ltln* (φ U ψ) = *ltl-to-ltln* φ U_n *ltl-to-ltln* ψ
| *ltl-to-ltln* (φ V ψ) = *ltl-to-ltln* φ V_n *ltl-to-ltln* ψ

lemma *ltl-to-ltln-on-ltl-pushneg-equiv*:
assumes φ = *ltl-pushneg* ψ
shows $\xi \models \varphi \longleftrightarrow \xi \models_n$ *ltl-to-ltln* φ
using *assms* **proof**(*induct* φ *arbitrary*: ξ ψ)
case *goal1* **show** *?case* **by** *auto*
next
case *goal2* **show** *?case* **by** *auto*
next
case *goal3* **show** *?case* **by** *auto*
next
case (*goal4* φ)
with *ltl-pushneg-neg-struct*[*of* ψ φ]
obtain q
where φ = *prop*(q)
by *auto*
thus *?case* **by** *auto*
next
case (*goal5 f g* ξ ψ)
hence *frml-eq*: *ltl-pushneg* ψ = *f and g* **by** *auto*
with *subformula-on-ltl-pushneg*[*of* - ψ]
obtain μ
where *f* = *ltl-pushneg* μ
by (*auto intro*: *subfrml.intros*)
moreover
from *frml-eq subformula-on-ltl-pushneg*[*of* - ψ]
obtain ν
where *g* = *ltl-pushneg* ν
by (*auto intro*: *subfrml.intros*)

```
    ultimately
    have ξ ⊨ f = ξ ⊨_n ltl-to-ltln f
     and ξ ⊨ g = ξ ⊨_n ltl-to-ltln g
    using goal5 by auto
    thus ?case by auto
next
  case (goal6 f g ξ ψ)
    hence frml-eq: ltl-pushneg ψ = f or g by auto
    with subformula-on-ltl-pushneg[of - ψ]
    obtain μ
     where f = ltl-pushneg μ
        by (auto intro: subfrml.intros)
    moreover
    from frml-eq subformula-on-ltl-pushneg[of - ψ]
    obtain ν
     where g = ltl-pushneg ν
        by (auto intro: subfrml.intros)
    ultimately
    have ξ ⊨ f = ξ ⊨_n ltl-to-ltln f
     and ξ ⊨ g = ξ ⊨_n ltl-to-ltln g
    using goal6 by auto
    thus ?case by auto
next
  case (goal7 φ ξ ψ)
    hence ltl-pushneg ψ = X φ by auto
    with subformula-on-ltl-pushneg[of φ ψ]
    obtain μ
     where φ = ltl-pushneg μ
        by (auto intro: subfrml.intros)
    hence suffix 1 ξ ⊨ φ = suffix 1 ξ ⊨_n ltl-to-ltln φ
    using goal7 by auto
    thus ?case by auto
next
  case (goal8 f g ξ ψ)
    hence frml-eq: ltl-pushneg ψ = f U g by auto
    with subformula-on-ltl-pushneg[of - ψ]
    obtain μ
     where f = ltl-pushneg μ
        by (auto intro: subfrml.intros)
    moreover
    from frml-eq subformula-on-ltl-pushneg[of - ψ]
    obtain ν
     where g = ltl-pushneg ν
        by (auto intro: subfrml.intros)
    ultimately
    have ∀ i. suffix i ξ ⊨ f = suffix i ξ ⊨_n ltl-to-ltln f
     and ∀ i. suffix i ξ ⊨ g = suffix i ξ ⊨_n ltl-to-ltln g
    using goal8 by auto
    thus ?case by auto
```

next
case (*goal9 f g ξ ψ*)
hence *frml-eq*: *ltl-pushneg ψ* = *f V g* **by** *auto*
with *subformula-on-ltl-pushneg*[*of* - *ψ*]
obtain *μ*
where *f* = *ltl-pushneg μ*
by (*auto intro*: *subfrml.intros*)
moreover
from *frml-eq subformula-on-ltl-pushneg*[*of* - *ψ*]
obtain *ν*
where *g* = *ltl-pushneg ν*
by (*auto intro*: *subfrml.intros*)
ultimately
have ∀ *i*. *suffix i ξ* ⊨ *f* = *suffix i ξ* ⊨$_n$ *ltl-to-ltln f*
and ∀ *i*. *suffix i ξ* ⊨ *g* = *suffix i ξ* ⊨$_n$ *ltl-to-ltln g*
using *goal9* **by** *auto*
thus *?case* **by** *auto*
qed

lemma *ltl-nnf-equiv*[*simp*]:
ξ ⊨$_n$ *ltl-to-ltln* (*ltl-pushneg ψ*) ⟷ *ξ* ⊨ *ψ*
using *sym*[*OF ltl-pushneg-equiv*] *ltl-to-ltln-on-ltl-pushneg-equiv* **by** *blast*

fun *subfrmlsn* :: *'a ltln* ⇒ *'a ltln set*
where
subfrmlsn (*μ and*$_n$ *ψ*) = {*μ and*$_n$ *ψ*} ∪ *subfrmlsn μ* ∪ *subfrmlsn ψ*
| *subfrmlsn* (*X*$_n$ *μ*) = {*X*$_n$ *μ*} ∪ *subfrmlsn μ*
| *subfrmlsn* (*μ U*$_n$ *ψ*) = {*μ U*$_n$ *ψ*} ∪ *subfrmlsn μ* ∪ *subfrmlsn ψ*
| *subfrmlsn* (*μ V*$_n$ *ψ*) = {*μ V*$_n$ *ψ*} ∪ *subfrmlsn μ* ∪ *subfrmlsn ψ*
| *subfrmlsn* (*μ or*$_n$ *ψ*) = {*μ or*$_n$ *ψ*} ∪ *subfrmlsn μ* ∪ *subfrmlsn ψ*
| *subfrmlsn x* = {*x*}

lemma *subfrmlsn-id*[*simp*]: *φ* ∈ *subfrmlsn φ* **by** (*induct φ*) *auto*
lemma *subfrmlsn-finite*: *finite* (*subfrmlsn φ*) **by** (*induct φ*) *auto*
lemma *subfrmlsn-subset*:*ψ* ∈ *subfrmlsn φ* ⟹ *subfrmlsn ψ* ⊆ *subfrmlsn φ*
by (*induct φ arbitrary*:*ψ*) *auto*

fun *size-frmln* :: *'a ltln* ⇒ *nat*
where
size-frmln (*φ and*$_n$ *ψ*) = *size-frmln φ* + *size-frmln ψ* + *1*
| *size-frmln* (*X*$_n$ *φ*) = *size-frmln φ* + *1*
| *size-frmln* (*φ U*$_n$ *ψ*) = *size-frmln φ* + *size-frmln ψ* + *1*
| *size-frmln* (*φ V*$_n$ *ψ*) = *size-frmln φ* + *size-frmln ψ* + *1*
| *size-frmln* (*φ or*$_n$ *ψ*) = *size-frmln φ* + *size-frmln ψ* + *1*
| *size-frmln* - = *1*

lemma *size-frmln-gt-zero*[*simp*]: *size-frmln φ* > *0* **by** (*induct φ*) *auto*

abbreviation
frmlset-sumn $\Phi \equiv$ *setsum size-frmln* Φ — FIXME: lemmas about this?

lemma *frmlset-sumn-diff-less*[*intro*!]:
assumes *finS*:*finite S*
and $A \neq \{\}$
and *subset*:$A \subseteq S$
shows *frmlset-sumn* $(S - A) <$ *frmlset-sumn S*
proof −
have *finA*: *finite A* **using** *assms* **by** (*rule-tac finite-subset*)
hence *frmlset-sumn A > 0* **using** *assms size-frmln-gt-zero* **by** (*induct rule*:*finite-induct*) *auto*
moreover
have *frmlset-sumn A* $\leq$ *frmlset-sumn S* **using** *assms size-frmln-gt-zero* **by** (*rule-tac setsum-mono2*) *auto*
ultimately show *?thesis* **using** *setsum-diff-nat*[*OF finA*, *of S size-frmln*] *assms*
by *auto*
qed

definition
frmln-props $\varphi \equiv \{p.\ prop_n(p) \in subfrmlsn\ \varphi \vee nprop_n(p) \in subfrmlsn\ \varphi\}$

lemma *ltln-expand-Until*:
$\xi \models_n \varphi\ U_n\ \psi = (\xi \models_n \psi\ or_n\ (\varphi\ and_n\ (X_n\ (\varphi\ U_n\ \psi))))$ (**is** *?lhs = ?rhs*)
proof
assume *?lhs*
then obtain *i*
where *psi-is*: *suffix i* $\xi \models_n \psi$
and *phi-is*: $\forall j<i.$ *suffix j* $\xi \models_n \varphi$ **by** *auto*
show *?rhs*
proof(*cases i*)
assume *i=0*
thus *?rhs* **using** *psi-is* **by** *auto*
next
fix *k*
assume *i-eq*: $i = Suc\ k$
with *phi-is* **have** $\xi \models_n \varphi$ **by** *auto*
moreover
have $\xi \models_n X_n\ (\varphi\ U_n\ \psi)$
using *psi-is phi-is i-eq* **by** *auto*
ultimately show *?rhs* **by** *auto*
qed
next
assume *rhs*: *?rhs*
show *?lhs*
proof(*cases* $\xi \models_n \psi$)
assume $\xi \models_n \psi$
hence *suffix 0* $\xi \models_n \psi$ **by** *auto*

```
    moreover
    have ∀j<0. suffix j ξ ⊨n φ by auto
    ultimately
    have ∃i. suffix i ξ ⊨n ψ
            ∧ (∀j<i. suffix j ξ ⊨n φ) by blast
    thus ?lhs by auto
  next
    assume ¬ ξ ⊨n ψ
    hence phi-is: ξ ⊨n φ
      and ξ ⊨n Xn (φ Un ψ) using rhs by auto
    then obtain i
         where psi-suc-is: suffix (Suc i) ξ ⊨n ψ
           and phi-suc-is: ∀j<i. suffix (Suc j) ξ ⊨n φ by auto
    have sbgoal: ∀j<(Suc i). suffix j ξ ⊨n φ
    proof(clarify)
      fix j
      assume j-less: j<Suc i
      show suffix j ξ ⊨n φ
      proof (cases j)
        assume j=0
        thus ?thesis using phi-is by auto
      next
        fix k
        assume j=Suc k
        thus ?thesis using j-less phi-suc-is by auto
      qed
    qed
    thus ?lhs using psi-suc-is phi-is by auto
  qed
qed

lemma ltln-expand-Release:
  ξ ⊨n φ Vn ψ = (ξ ⊨n ψ andn (φ orn (Xn (φ Vn ψ)))) (is ?lhs = ?rhs)
proof
  assume lhs: ?lhs
  hence psi-is: ξ ⊨n ψ by force

  have ⋀i. ⟦¬ ξ ⊨n φ; ¬ suffix (Suc i) ξ ⊨n ψ⟧
          ⟹ (∃j<i. suffix (Suc j) ξ ⊨n φ)
  proof −
    fix i
    assume phi-nis: ¬ ξ ⊨n φ
       and ¬ suffix (Suc i) ξ ⊨n ψ
    then obtain j
         where j<Suc i
           and suffix j ξ ⊨n φ using lhs by auto
    hence j − 1 < i ∧ suffix (Suc (j − 1)) ξ ⊨n φ
    using phi-nis by (cases j) auto
    thus ∃j<i. suffix (Suc j) ξ ⊨n φ by auto
```

```
  qed
  thus ?rhs using psi-is by auto
next
  assume rhs: ?rhs
  hence psi-is: ξ ⊨n ψ by auto

  show ?lhs
  proof(cases ξ ⊨n φ)
    assume ξ ⊨n φ
    thus ?thesis using psi-is by force
  next
    assume phi-nis: ¬ ξ ⊨n φ

    hence ∀ i. suffix (Suc i) ξ ⊨n ψ
              ∨ (∃ j<Suc i. suffix j ξ ⊨n φ)
    using rhs by auto

    have ⋀i. ¬ suffix i ξ ⊨n ψ
            ⟹ (∃ j<i. suffix j ξ ⊨n φ)
    proof −
      fix i
      assume psi-suf-nis: ¬ suffix i ξ ⊨n ψ
      show ∃ j<i. suffix j ξ ⊨n φ
      proof(cases i)
        assume i=0
        with psi-suf-nis psi-is show ?thesis by auto
      next
        fix k
        assume i-eq: i=Suc k
        with psi-suf-nis rhs show ?thesis by force
      qed
    qed
    thus ?thesis by auto
  qed
qed

lemma ltln-Release-alterdef:
  ξ ⊨n φ V n ψ ⟷ ξ ⊨n (□n ψ) orn (ψ U n (φ andn ψ)) (is ?lhs = ?rhs)
proof
  assume ?lhs
  { assume ¬ (∀ i. suffix i ξ ⊨n ψ)
    then obtain i where ψ-neq: ¬ suffix i ξ ⊨n ψ by auto
    let ?k = LEAST i. ¬ suffix i ξ ⊨n ψ
    from ψ-neq ‹?lhs› have ∀ j<?k. suffix j ξ ⊨n ψ by (metis not-less-Least)
    moreover
    have ¬ suffix ?k ξ ⊨n ψ by (rule LeastI, rule ψ-neq)
    moreover then obtain j where j<?k and suffix j ξ ⊨n φ using ‹?lhs› by
auto
    ultimately have ξ ⊨n ψ U n (φ andn ψ) by auto }
```

with ‹*?lhs*› **show** *?rhs* **by** *auto*
next
assume *?rhs*
{ assume $\neg\ \xi \models_n \Box_n\ \psi$
with ‹*?rhs*› **obtain** *i* **where** *suffix i* $\xi \models_n \varphi$ *and*$_n$ ψ **and** $\forall j{<}i.$ *suffix j* $\xi \models_n \psi$ **by** *auto*
hence *?lhs* **by** (*auto*, *metis nat-neq-iff*) **}**
thus *?lhs* **using** ‹*?rhs*› **by** *auto*
qed

end
end

3 Rewriting LTL formulas

theory *LTL-Rewrite*
imports
LTL
begin

context begin interpretation *LTL-Syntax* **.**

inductive-set *ltln-pure-eventual-frmls* :: *'a ltln set*
where
$\Diamond_n\ \varphi \in$ *ltln-pure-eventual-frmls*
| ⟦ $\nu \in$ *ltln-pure-eventual-frmls*; $\mu \in$ *ltln-pure-eventual-frmls* ⟧
$\Longrightarrow \nu$ *and*$_n$ $\mu \in$ *ltln-pure-eventual-frmls*
| ⟦ $\nu \in$ *ltln-pure-eventual-frmls*; $\mu \in$ *ltln-pure-eventual-frmls* ⟧
$\Longrightarrow \nu$ *or*$_n$ $\mu \in$ *ltln-pure-eventual-frmls*
| ⟦ $\nu \in$ *ltln-pure-eventual-frmls*; $\mu \in$ *ltln-pure-eventual-frmls* ⟧
$\Longrightarrow \nu\ U_n\ \mu \in$ *ltln-pure-eventual-frmls*
| ⟦ $\nu \in$ *ltln-pure-eventual-frmls*; $\mu \in$ *ltln-pure-eventual-frmls* ⟧
$\Longrightarrow \nu\ V_n\ \mu \in$ *ltln-pure-eventual-frmls*

theorem *ltln-pure-eventual-frmls-equiv*:
assumes $\psi \in$ *ltln-pure-eventual-frmls*
shows $\xi \models_n \varphi\ U_n\ \psi \longleftrightarrow \xi \models_n \psi$
using *assms* **proof** (*induct* ψ *arbitrary*:ξ φ)
case *goal1*
thus *?case*
by (*auto*, *metis comm-semiring-1-class.normalizing-semiring-rules*(*5*)
less-nat-zero-code)
next
case (*goal2* - - ξ φ) **show** *?case* **using** *goal2*(*2*)[*of* ξ φ] *goal2*(*4*)[*of* ξ φ]
by (*auto*, *metis less-nat-zero-code suffix-0*)
next
case (*goal3* - - ξ) **show** *?case*
using *goal3*(*2*)[*of* ξ φ] *goal3*(*4*)[*of* ξ φ] **by** *auto*

```
next
  case (goal4 ν μ ξ)
    let ?ψ = ν U_n μ
    { assume ξ ⊨_n φ U_n ?ψ
      then obtain i where suffix i ξ ⊨_n ?ψ and ∀j<i. suffix j ξ ⊨_n φ
        by auto
      moreover with goal4(4)[of suffix i ξ ν] have suffix i ξ ⊨_n μ
        by auto
      ultimately have ξ ⊨_n ?ψ using goal4(4)[of ξ φ] goal4(4)[of ξ ν]
        by auto }
    moreover
    { assume ξ ⊨_n ?ψ
      with goal4 have ξ ⊨_n φ U_n μ by auto
      then obtain i where suffix i ξ ⊨_n μ and ∀j<i. suffix j ξ ⊨_n φ
        by auto
      moreover with goal4(4)[of suffix i ξ ν] have suffix i ξ ⊨_n ν U_n μ
        by auto
      ultimately have ξ ⊨_n φ U_n ?ψ by auto }
    ultimately show ?case by fast
next
  case (goal5 ν μ)
    let ?ψ = ν V_n μ
    { assume ξ ⊨_n φ U_n ?ψ
      then obtain i where
        V-suf-i: suffix i ξ ⊨_n ν V_n μ
        and phi-all-less-i: ∀j<i. suffix j ξ ⊨_n φ
        unfolding ltln-Release-alterdef[symmetric] by auto
      hence μ-suf-i: suffix i ξ ⊨_n μ
        by (metis ltln-expand-Release ltln-semantics.simps(5))
      have μ-less-i: ∀j<i. suffix j ξ ⊨_n μ
      proof(clarify)
        fix k
        assume k<i
        hence suffix (i−k) (suffix k ξ) ⊨_n μ
          and ∀j<i−k. suffix j (suffix k ξ) ⊨_n φ
        using V-suf-i phi-all-less-i μ-suf-i by auto
        thus suffix k ξ ⊨_n μ using goal5(4)[of suffix k ξ φ] by auto
      qed
      have ξ ⊨_n ?ψ
      proof −
        { fix i'
          { assume i' < i
            hence suffix i' ξ ⊨_n μ using μ-less-i by auto }
          moreover
          { assume i' ≥ i
            then obtain i'' where i'-eq: i' = i + i'' and i'≥i''
              by (metis Nat.diff-le-self le-add-diff-inverse2 add.commute)
            hence suffix i' ξ ⊨_n μ ∨ (∃j<i'. suffix j ξ ⊨_n ν)
              using V-suf-i by auto }
```

```
      ultimately have suffix i' ξ ⊨n μ ∨ (∃j<i'. suffix j ξ ⊨n ν)
        by (metis linorder-not-less) }
     thus ?thesis by auto
   qed }
  moreover
  { assume ξ ⊨n ?ψ
   hence suffix 0 ξ ⊨n ?ψ ∧ (∀j<0. suffix j ξ ⊨n φ) by auto
   hence ξ ⊨n φ U n ?ψ unfolding ltln-semantics.simps by blast }
  ultimately show ?case by fast
qed

corollary ltln-pure-eventual-frmls-equiv-diamond:
 assumes ψ ∈ ltln-pure-eventual-frmls
 shows ξ ⊨n ◇n ψ ⟷ ξ ⊨n ψ
by (rule ltln-pure-eventual-frmls-equiv[OF assms])


inductive-set ltln-pure-universal-frmls :: 'a ltln set
where
 □n φ ∈ ltln-pure-universal-frmls
| ⟦ ν ∈ ltln-pure-universal-frmls; μ ∈ ltln-pure-universal-frmls ⟧
     ⟹ ν and n μ ∈ ltln-pure-universal-frmls
| ⟦ ν ∈ ltln-pure-universal-frmls; μ ∈ ltln-pure-universal-frmls ⟧
     ⟹ ν or n μ ∈ ltln-pure-universal-frmls
| ⟦ ν ∈ ltln-pure-universal-frmls; μ ∈ ltln-pure-universal-frmls ⟧
     ⟹ ν U n μ ∈ ltln-pure-universal-frmls
| ⟦ ν ∈ ltln-pure-universal-frmls; μ ∈ ltln-pure-universal-frmls ⟧
     ⟹ ν V n μ ∈ ltln-pure-universal-frmls

theorem ltln-pure-universal-frmls-equiv:
 assumes ψ ∈ ltln-pure-universal-frmls
 shows ξ ⊨n φ V n ψ ⟷ ξ ⊨n ψ
using assms proof (induct ψ arbitrary:ξ φ)
 case goal1
 thus ?case
   by (auto, metis comm-semiring-1-class.normalizing-semiring-rules(5)
     less-nat-zero-code)
next
 case (goal2 - - ξ φ)
 show ?case
   using goal2(2)[of ξ φ] goal2(4)[of ξ φ] by auto
next
 case (goal3 - - ξ) show ?case
   using goal3(2)[of ξ φ] goal3(4)[of ξ φ]
   by (auto, metis less-nat-zero-code suffix-0)
next
 case (goal4 ν μ ξ)
   let ?ψ = ν U n μ
   { assume assm: ξ ⊨n φ V n ?ψ
```

```
  { assume ξ ⊨n □n ?ψ
    hence suffix 0 ξ ⊨n ?ψ
      by (metis ltln-semantics.simps(9) ltln-semantics.simps(2))
    hence ξ ⊨n ?ψ by auto }
  moreover
  { assume ¬ ξ ⊨n □n ?ψ
    hence  ξ ⊨n ?ψ Un (φ andn ?ψ)
      using assm ltln-Release-alterdef[of ξ φ ?ψ]
      by (metis ltln-semantics.simps(6))
    then obtain i
      where suffix i ξ ⊨n φ andn ?ψ and ∀j<i. suffix j ξ ⊨n ?ψ
      by auto
    hence ξ ⊨n ?ψ
      by (cases i=0) (metis suffix-0 ltln-semantics.simps(5)
        neq0-conv suffix-0)+ }
  ultimately have ξ ⊨n ?ψ by fast }
moreover
{ assume ξ ⊨n ?ψ
  then obtain i where
    μ-suf-i: suffix i ξ ⊨n μ
    and ν-less-i: ∀j<i. suffix j ξ ⊨n ν
    by auto
  with goal4(4)[of suffix i ξ] μ-suf-i
  have suffix i ξ ⊨n □n μ orn (μ Un (φ andn μ))
    using ltln-Release-alterdef[of suffix i ξ φ μ] by auto
  moreover
  { assume suffix i ξ ⊨n □n μ
    hence ψ-suf-i: suffix i ξ ⊨n φ Vn ?ψ
      by auto (metis ac-simps ltln-expand-Until ltln-semantics.simps)
    from ν-less-i have ψ-less-i: ∀j<i. suffix j ξ ⊨n ?ψ
    proof(clarify)
      case (goal1 j)
        then obtain j' where i = j + j' by (metis less-imp-add-positive)
        hence suffix j' (suffix j ξ) ⊨n μ
          and ∀k<j'. suffix k (suffix j ξ) ⊨n ν
        using μ-suf-i by auto (metis ν-less-i ‹i = j + j'›
          add-less-cancel-right add.commute)
        thus ?case by auto
    qed
    have ξ ⊨n φ Vn ?ψ
    proof −
      { fix k
        { assume k < i
          with ψ-less-i have suffix k ξ ⊨n ?ψ by auto }
        moreover
        { assume k ≥ i
          then obtain i' where k = i + i' by (metis Nat.le-iff-add)
          with ψ-suf-i have suffix k ξ ⊨n ?ψ ∨ (∃j<k. suffix j ξ ⊨n φ)
            by auto }
```

```
        ultimately have suffix k ξ ⊨n ?ψ ∨ (∃j<k. suffix j ξ ⊨n φ)
          by (metis linorder-not-less) }
       thus ?thesis by auto
     qed }
   moreover
   { assume suffix i ξ ⊨n μ Un (φ andn μ)
     then obtain k where suffix (i+k) ξ ⊨n φ
                   and suffix (i+k) ξ ⊨n μ
                   and ∀j<k. suffix j (suffix i ξ) ⊨n μ by auto
     hence ∀j≤i+k. suffix j ξ ⊨n ?ψ
     proof(clarify)
       case (goal1 j)
         { assume j < i
           then obtain j' where i = j + j' by (metis less-imp-add-positive)
           with μ-suf-i ν-less-i have ?case by auto }
         moreover
         { assume j ≥ i
           then obtain i' where j = i + i' by (metis Nat.le-iff-add)
           with goal1 have ?case
             by (auto, metis (full-types) add.comm-neutral add-Suc-right
               le-neq-implies-less less-nat-zero-code) }
         ultimately show ?case by (metis less-or-eq-imp-le linorder-neqE-nat)
     qed
     with ‹suffix (i+k) ξ ⊨n φ› have ξ ⊨n φ Vn ?ψ
       by (auto, metis linorder-not-less) }
   ultimately have ξ ⊨n φ Vn ?ψ by auto }
 ultimately show ?case by fast
next
 case (goal5 ν μ ξ)
   let ?ψ = ν Vn μ
   { assume ξ ⊨n φ Vn ?ψ
     moreover
     { assume ξ ⊨n □n ?ψ
       hence ξ ⊨n ?ψ
         by (metis goal5(4) ltln-semantics.simps(2) ltln-semantics.simps(9)) }
     moreover
     { assume ξ ⊨n ?ψ Un (φ andn ?ψ)
       then obtain i where suffix i ξ ⊨n ?ψ and ∀j<i. suffix j ξ ⊨n ?ψ
         by auto
       hence ξ ⊨n ?ψ by (cases i=0) (auto, metis suffix-0 suffix-suffix) }
     ultimately have ξ ⊨n ?ψ using ltln-Release-alterdef[of ξ φ ?ψ] by auto }
   moreover
   { assume ξ ⊨n ?ψ
     { assume ξ ⊨n □n μ
       hence ξ ⊨n φ Vn ?ψ by auto }
     moreover
     { assume assm: ¬ ξ ⊨n □n μ
       hence ξ ⊨n μ Un (ν andn μ)
         using ltln-Release-alterdef[of ξ ν μ] ‹ξ ⊨n ?ψ› by auto
```

```
      hence ξ ⊨n μ using ‹ξ ⊨n ?ψ›  by (auto, metis calculation(1) goal5(4))
      with goal5(4)[of ξ φ] have ξ ⊨n φ Vn μ by auto
      hence ξ ⊨n μ Un (φ andn μ)
        using assm ltln-Release-alterdef[of ξ φ μ] by auto
      then obtain i
        where suffix i ξ ⊨n φ andn μ and ∀j<i. suffix j ξ ⊨n μ
        by auto
      moreover hence ∀j≤i. suffix j ξ ⊨n ν Vn μ
        by (metis ‹ξ ⊨n μ› assm goal5(4))
      ultimately have ξ ⊨n φ Vn ?ψ by (auto, metis linorder-not-le) }
    ultimately have ξ ⊨n φ Vn ?ψ by fast }
  ultimately show ?case by fast
qed
```

Some simple rewrite rules

```
fun ltln-rewrite-step :: 'a ltln ⇒ 'a ltln
where
  ltln-rewrite-step (- Un truen) = truen
| ltln-rewrite-step (- Vn falsen) = falsen
| ltln-rewrite-step (truen Un (- Un μ)) = truen Un μ
| ltln-rewrite-step (falsen Vn (- Vn μ)) = falsen Vn μ
| ltln-rewrite-step ψ = (case ψ of
    φ Un φ' ⇒
      if φ = φ' then φ
      else if φ' ∈ ltln-pure-eventual-frmls then φ'
      else ψ
  | φ Vn φ' ⇒
      if φ = φ' then φ
      else if φ' ∈ ltln-pure-universal-frmls then φ'
      else ψ
  | (φ Un μ) andn (ν Un μ') ⇒ if μ = μ' then (φ andn ν) Un μ else ψ
  | (φ Un ν) orn (φ' Un μ) ⇒ if φ = φ' then φ Un (ν orn μ) else ψ
  | (φ Vn ν) andn (φ' Vn μ) ⇒ if φ = φ' then φ Vn (ν andn μ) else ψ
  | (φ Vn μ) orn (ν Vn μ') ⇒ if μ = μ' then (φ orn ν) Vn μ else ψ
  | - ⇒ ψ)

lemma ltln-rewrite-step--size-less:
  assumes ltln-rewrite-step ψ ≠ ψ
  shows size (ltln-rewrite-step ψ) < size ψ
  using assms proof (cases ψ)
  case (goal8 ν μ)
  thus ?case
    by (cases μ, cases ν) (auto split:ltln.split,
      metis+, cases ν, auto split:ltln.split, metis+)
next
  case (goal9 ν μ)
  thus ?case
    by (cases μ, cases ν) (auto split:ltln.split,
      metis+, cases ν, auto split:ltln.split, metis+)
```

qed (*auto split*:*ltln.split*)

lemma *ltln-rewrite-step--size-leq*:
size (*ltln-rewrite-step* ψ) $\leq$ *size* ψ
using *ltln-rewrite-step--size-less*[*of* ψ]
by (*cases ltln-rewrite-step* $\psi = \psi$) *auto*

theorem *ltln-rewrite-step--equiv*:
$\xi \models_n$ *ltln-rewrite-step* $\psi \longleftrightarrow \xi \models_n \psi$
proof (*cases* ψ)
case (*goal5* ν μ) **thus** *?case*
proof (*cases* ν)
case *goal8* **thus** *?case*
proof (*cases* μ)
case *goal8* **thus** *?case* **by** (*auto*, *metis nat-neq-iff order-less-trans*)
qed *auto*
qed (*auto split*:*ltln.split*)
next
case (*goal6* ν μ) **thus** *?case*
proof (*cases* ν)
case *goal9* **thus** *?case*
proof (*cases* μ)
case *goal9* **thus** *?case* **by** (*auto*, *metis nat-neq-iff order-less-trans*)
qed *auto*
qed (*auto split*:*ltln.split*)
next
case (*goal8* ν μ) **thus** *?case*
proof(*cases* $\mu \neq true_n \wedge \neg\ (\nu = true_n \wedge (\exists \nu'\ \mu'.\ \mu = \nu'\ U_n\ \mu')) \wedge \nu \neq \mu$)
case *goal1*
hence $\xi \models_n$ (*if* $\mu \in$ *ltln-pure-eventual-frmls then* μ *else* ψ) $\longleftrightarrow \xi \models_n \psi$
using *ltln-pure-eventual-frmls-equiv* **by** *auto*
thus *?case* **using** *goal1* **by** (*cases* μ) (*cases* ν, *auto split*:*ltln.split*)+
next
case *goal2*
thus *?case*
by (*cases* μ, *auto split*:*ltln.split*) (*metis less-nat-zero-code neq0-conv*
suffix-0 add.comm-neutral add-0)+
qed
next
case (*goal9* ν μ) **thus** *?case*
proof(*cases* $\mu \neq false_n \wedge \neg\ (\nu = false_n \wedge (\exists \nu'\ \mu'.\ \mu = \nu'\ V_n\ \mu')) \wedge \nu \neq \mu$)
case *goal1*
hence $\xi \models_n$ (*if* $\mu \in$ *ltln-pure-universal-frmls then* μ *else* ψ) $\longleftrightarrow \xi \models_n \psi$
using *ltln-pure-universal-frmls-equiv* **by** *auto*
thus *?case* **using** *goal1* **by** (*cases* μ) (*cases* ν, *auto split*:*ltln.split*)+
next
case *goal2*
thus *?case*

```
    by (cases μ, auto split:ltln.split) (metis less-nat-zero-code neq0-conv
      suffix-0 add.comm-neutral add-0)+
  qed
qed (auto split:ltln.split)

function ltln-rewrite-rec
where
  ltln-rewrite-rec ψ = (case ltln-rewrite-step ψ of
    ν and_n μ ⇒ (ltln-rewrite-rec ν) and_n (ltln-rewrite-rec μ)
  | ν or_n μ ⇒ (ltln-rewrite-rec ν) or_n (ltln-rewrite-rec μ)
  | X_n ν ⇒ X_n (ltln-rewrite-rec ν)
  | ν U_n μ ⇒ (ltln-rewrite-rec ν) U_n (ltln-rewrite-rec μ)
  | ν V_n μ ⇒ (ltln-rewrite-rec ν) V_n (ltln-rewrite-rec μ)
  | φ ⇒ φ)
by pat-completeness auto
termination proof −
  {
    fix ψ φ :: 'a ltln and thesis
    assume ltln-rewrite-step ψ = φ
    thm ltln-rewrite-step--size-leq
    moreover assume ⟦ltln-rewrite-step ψ = φ;
      size (local.ltln-rewrite-step ψ) ≤ size ψ⟧ ⟹ thesis
    ultimately have thesis using ltln-rewrite-step--size-leq[of ψ]
      by blast
  } note AUX=this
  show ?thesis
    apply (relation inv-image less-than size)
    apply simp-all
    apply (erule AUX, simp)+
    done
qed

declare ltln-rewrite-rec.simps [simp del]

lemma ltln-rewrite-rec--size-less:
  assumes ltln-rewrite-rec ψ ≠ ψ
  shows size (ltln-rewrite-rec ψ) < size ψ
using assms proof (induct ltln-rewrite-rec ψ arbitrary:ψ)
  case goal1 thus ?case
    using ltln-rewrite-step--size-less[of ψ] ltln-rewrite-rec.simps[of ψ]
    by (cases ltln-rewrite-step ψ) auto
next
  case goal2 thus ?case
    using ltln-rewrite-step--size-less[of ψ] ltln-rewrite-rec.simps[of ψ]
    by (cases ltln-rewrite-step ψ) auto
next
  case goal3 thus ?case
    using ltln-rewrite-step--size-less[of ψ] ltln-rewrite-rec.simps[of ψ]
    by (cases ltln-rewrite-step ψ) auto
```

next
case *goal4* **thus** *?case*
using *ltln-rewrite-step--size-less*[*of* ψ] *ltln-rewrite-rec.simps*[*of* ψ]
by (*cases ltln-rewrite-step* ψ) *auto*
next
case (*goal5* ν μ) **thus** *?case* **using** *ltln-rewrite-step--size-less*[*of* ψ]
proof (*cases ltln-rewrite-step* ψ)
case (*goal5* ν' μ') **thus** *?case*
unfolding *goal5*(*6*) *ltln-rewrite-rec.simps*[*of* ψ]
apply (*cases* ν' and_n $\mu' = \psi$)
by *fastforce*+
qed (*auto simp add*: *ltln-rewrite-rec.simps*[*of* ψ])
next
case (*goal6* ν μ) **thus** *?case* **using** *ltln-rewrite-step--size-less*[*of* ψ]
proof (*cases ltln-rewrite-step* ψ)
case (*goal6* ν' μ') **thus** *?case*
unfolding *goal6*(*6*) *ltln-rewrite-rec.simps*[*of* ψ]
by (*cases* ν' or_n $\mu' = \psi$) (*fastforce*+)
qed (*auto simp add*: *ltln-rewrite-rec.simps*[*of* ψ])
next
case (*goal7* ν) **thus** *?case* **using** *ltln-rewrite-step--size-less*[*of* ψ]
proof (*cases ltln-rewrite-step* ψ)
case (*goal7* ν') **thus** *?case* **unfolding** *goal7*(*5*) *ltln-rewrite-rec.simps*[*of* ψ]
by (*cases* X_n $\nu' = \psi$) (*fastforce*+)
qed (*auto simp add*: *ltln-rewrite-rec.simps*[*of* ψ])
next
case (*goal8* ν μ) **thus** *?case* **using** *ltln-rewrite-step--size-less*[*of* ψ]
proof (*cases ltln-rewrite-step* ψ)
case (*goal8* ν' μ')
thus *?case* **unfolding** *goal8*(*6*) *ltln-rewrite-rec.simps*[*of* ψ]
by (*cases* ν' U_n $\mu' = \psi$) (*fastforce*+)
qed (*auto simp add*: *ltln-rewrite-rec.simps*[*of* ψ])
next
case (*goal9* ν μ) **thus** *?case* **using** *ltln-rewrite-step--size-less*[*of* ψ]
proof (*cases ltln-rewrite-step* ψ)
case (*goal9* ν' μ')
thus *?case* **unfolding** *goal9*(*6*) *ltln-rewrite-rec.simps*[*of* ψ]
by (*cases* ν' V_n $\mu' = \psi$) (*fastforce*+)
qed (*auto simp add*: *ltln-rewrite-rec.simps*[*of* ψ])
qed

lemma *ltln-rewrite-rec--size-leq*:
size (*ltln-rewrite-rec* ψ) $\leq$ *size* ψ
using *ltln-rewrite-rec--size-less*[*of* ψ] **by** (*cases ltln-rewrite-rec* $\psi = \psi$) *auto*

theorem *ltln-rewrite-rec--equiv*:
$\xi \models_n$ *ltln-rewrite-rec* $\psi \longleftrightarrow \xi \models_n \psi$
proof (*induct ltln-rewrite-rec* ψ *arbitrary*:ξ ψ)

```
  case goal1
    thus ?case
      using ltln-rewrite-step--equiv[of - ψ] ltln-rewrite-rec.simps[of ψ]
    by (cases ltln-rewrite-step ψ) auto
next
  case goal2
    thus ?case
      using ltln-rewrite-step--equiv[of - ψ] ltln-rewrite-rec.simps[of ψ]
    by (cases ltln-rewrite-step ψ) auto
next
  case goal3
    thus ?case
      using ltln-rewrite-step--equiv[of - ψ] ltln-rewrite-rec.simps[of ψ]
    by (cases ltln-rewrite-step ψ) auto
next
  case goal4
    thus ?case
      using ltln-rewrite-step--equiv[of - ψ] ltln-rewrite-rec.simps[of ψ]
    by (cases ltln-rewrite-step ψ) auto
next
  case goal5
    thus ?case
      using ltln-rewrite-step--equiv[of - ψ] ltln-rewrite-rec.simps[of ψ]
    by (cases ltln-rewrite-step ψ) auto
next
  case goal6
    thus ?case
      using ltln-rewrite-step--equiv[of - ψ] ltln-rewrite-rec.simps[of ψ]
    by (cases ltln-rewrite-step ψ) auto
next
  case goal7
    thus ?case
      using ltln-rewrite-step--equiv[of - ψ] ltln-rewrite-rec.simps[of ψ]
    by (cases ltln-rewrite-step ψ) auto
next
  case goal8
    thus ?case
      using ltln-rewrite-step--equiv[of - ψ] ltln-rewrite-rec.simps[of ψ]
    by (cases ltln-rewrite-step ψ) auto
next
  case goal9
    thus ?case
      using ltln-rewrite-step--equiv[of - ψ] ltln-rewrite-rec.simps[of ψ]
    by (cases ltln-rewrite-step ψ) auto
qed


function ltln-rewrite
where
```

```
  ltln-rewrite ψ
    = (let φ = ltln-rewrite-rec ψ in
         if φ ≠ ψ then ltln-rewrite φ else ψ)
by pat-completeness auto
termination
proof(relation inv-image less-than size)
  case goal1 thus ?case by auto
next
  case (goal2 ψ) thus ?case using ltln-rewrite-rec--size-less[of ψ] by auto
qed

declare ltln-rewrite.simps [simp del]


lemma ltln-rewrite--size-less:
  assumes ltln-rewrite ψ ≠ ψ
  shows size (ltln-rewrite ψ) < size ψ
using assms proof (induct ψ rule: ltln-rewrite.induct)
  case (goal1 ψ)
    { assume ltln-rewrite-rec ψ = ψ
      hence ?case using goal1(2) unfolding ltln-rewrite.simps[of ψ] by auto }
    moreover
    { assume assm: ltln-rewrite-rec ψ ≠ ψ
      hence rw-ψ-eq: ltln-rewrite ψ = ltln-rewrite (ltln-rewrite-rec ψ)
        unfolding ltln-rewrite.simps[of ψ] by auto
      from goal1(1)[OF refl assm, folded rw-ψ-eq]
        and ltln-rewrite-rec--size-less[OF assm]
      have ?case by (cases ltln-rewrite ψ = ltln-rewrite-rec ψ) auto }
    ultimately show ?case by fast
qed

lemma ltln-rewrite--size-leq:
  size (ltln-rewrite ψ) ≤ size ψ
using ltln-rewrite--size-less[of ψ] by (cases ltln-rewrite ψ = ψ) auto


lemma ltln-rewrite--equiv:
  ξ ⊨n ltln-rewrite ψ ⟷ ξ ⊨n ψ
proof (induct ψ rule: ltln-rewrite.induct)
  case (goal1 ψ)
    show ?case using goal1[OF refl] ltln-rewrite-rec--equiv[of ξ ψ]
    unfolding ltln-rewrite.simps[of ψ] by (cases ltln-rewrite-rec ψ = ψ) auto
qed


fun ltln-pure-eventual-frmls-impl
where
  ltln-pure-eventual-frmls-impl (◇n φ) = True
| ltln-pure-eventual-frmls-impl (ν andn μ)
```

```
  = (ltln-pure-eventual-frmls-impl ν ∧ ltln-pure-eventual-frmls-impl μ)
| ltln-pure-eventual-frmls-impl (ν or_n μ)
  = (ltln-pure-eventual-frmls-impl ν ∧ ltln-pure-eventual-frmls-impl μ)
| ltln-pure-eventual-frmls-impl (ν U_n μ)
  = (ltln-pure-eventual-frmls-impl ν ∧ ltln-pure-eventual-frmls-impl μ)
| ltln-pure-eventual-frmls-impl (ν V_n μ)
  = (ltln-pure-eventual-frmls-impl ν ∧ ltln-pure-eventual-frmls-impl μ)
| ltln-pure-eventual-frmls-impl - = False

lemma ltln-pure-eventual-frmls-unfold[code-unfold]:
  φ ∈ ltln-pure-eventual-frmls ⟷ ltln-pure-eventual-frmls-impl φ
  (is ?lhs = ?rhs)
proof
  assume ?lhs
  thus ?rhs
  proof (induct rule: ltln-pure-eventual-frmls.induct)
    case goal4 thus ?case by (cases ν = true_n) (cases ν, auto)+
  qed auto
next
  assume ?rhs
  thus ?lhs
  proof (induct φ)
    case (goal8 ν μ)
    thus ?case
      by (cases ν = true_n) (cases ν,
        auto intro: ltln-pure-eventual-frmls.intros)+
  qed (auto intro: ltln-pure-eventual-frmls.intros)
qed

fun ltln-pure-universal-frmls-impl
where
  ltln-pure-universal-frmls-impl (□_n φ) = True
| ltln-pure-universal-frmls-impl (ν and_n μ)
  = (ltln-pure-universal-frmls-impl ν ∧ ltln-pure-universal-frmls-impl μ)
| ltln-pure-universal-frmls-impl (ν or_n μ)
  = (ltln-pure-universal-frmls-impl ν ∧ ltln-pure-universal-frmls-impl μ)
| ltln-pure-universal-frmls-impl (ν U_n μ)
  = (ltln-pure-universal-frmls-impl ν ∧ ltln-pure-universal-frmls-impl μ)
| ltln-pure-universal-frmls-impl (ν V_n μ)
  = (ltln-pure-universal-frmls-impl ν ∧ ltln-pure-universal-frmls-impl μ)
| ltln-pure-universal-frmls-impl - = False

lemma ltln-pure-universal-frmls-unfold[code-unfold]:
  φ ∈ ltln-pure-universal-frmls ⟷ ltln-pure-universal-frmls-impl φ
  (is ?lhs = ?rhs)
proof
  assume ?lhs
  thus ?rhs
  proof (induct rule: ltln-pure-universal-frmls.induct)
```

```
    case goal5 thus ?case by (cases ν = false_n) (cases ν, auto)+
  qed auto
next
  assume ?rhs
  thus ?lhs
  proof (induct φ)
    case (goal9 ν μ)
    thus ?case
      by (cases ν = false_n) (cases ν,
        auto intro: ltln-pure-universal-frmls.intros)+
  qed (auto intro: ltln-pure-universal-frmls.intros)
qed

definition
  ltln-rewrite-step-impl ψ ≡ case ψ of
    ν U_n μ ⇒ if μ = true_n then true_n
              else (
                case (ν, μ) of
                  (true_n, - U_n μ') ⇒ true_n U_n μ'
                | - ⇒ if ν = μ then ν
                      else if μ∈ltln-pure-eventual-frmls then μ
                      else ψ)
  | ν V_n μ ⇒ if μ = false_n then false_n
              else (
                case (ν, μ) of
                  (false_n, - V_n μ') ⇒ false_n V_n μ'
                | - ⇒ if ν = μ then ν
                      else if μ∈ltln-pure-universal-frmls then μ
                      else ψ)
  | ψ1 and_n ψ2 ⇒ (
      case (ψ1, ψ2) of
        (φ U_n μ, ν U_n μ') ⇒ if μ = μ' then (φ and_n ν) U_n μ else ψ
      | (φ V_n ν, φ' V_n μ) ⇒ if φ = φ' then φ V_n (ν and_n μ) else ψ
      | - ⇒ ψ)
  | ψ1 or_n ψ2 ⇒ (
      case (ψ1, ψ2) of
        (φ U_n ν, φ' U_n μ) ⇒ if φ = φ' then φ U_n (ν or_n μ) else ψ
      | (φ V_n μ, ν V_n μ') ⇒ if μ = μ' then (φ or_n ν) V_n μ else ψ
      | - ⇒ ψ)
  | - ⇒ ψ

lemma ltln-rewrite-step-unfold[code-unfold]:
  ltln-rewrite-step = ltln-rewrite-step-impl
proof −
  { fix ψ :: 'a ltln
    have ltln-rewrite-step ψ = ltln-rewrite-step-impl ψ
    unfolding ltln-rewrite-step-impl-def by (cases ψ) (auto split:ltln.split) }
  thus ?thesis by auto
qed
```

end

end

4 Stutter Invariance of next-free LTL Formula

theory *LTL-Stutter*
imports *LTL ../Stuttering-Equivalence/PLTL*
begin

This theory builds on the AFP-entry by Stephan Merz

Get rid of overlapping notation

no-notation *PLTL.holds-of* (- ⊨ - [*70*,*70*] *40*)

hide-const (**open**) *PLTL.false PLTL.atom PLTL.implies PLTL.next PLTL.until*
hide-const (**open**) *PLTL.not PLTL.true PLTL.or PLTL.and*
hide-const (**open**) *PLTL.eventually PLTL.always*

no-notation *Samplers.suffix* (-[- ..] [*80*, *15*] *80*)
hide-const (**open**) *Samplers.suffix*
hide-fact (**open**) *Samplers.suffix-def*

lemma *PLTL-suffix-cnv*[*simp*]: *Samplers.suffix* = (λ*w i*. *suffix i w*)
apply (*intro ext*)
unfolding *Samplers.suffix-def Words.suffix-def* **..**

hide-const (**open**) *PLTL.next-free*

primrec *ltl-next-free* :: *'a ltl* ⇒ *bool* **where**
ltl-next-free LTLTrue ⟷ *True*
| *ltl-next-free LTLFalse* ⟷ *True*
| *ltl-next-free* (*LTLProp* -) ⟷ *True*
| *ltl-next-free* (*LTLNeg* φ) ⟷ *ltl-next-free* φ
| *ltl-next-free* (*LTLAnd* φ ψ) ⟷ *ltl-next-free* φ ∧ *ltl-next-free* ψ
| *ltl-next-free* (*LTLOr* φ ψ) ⟷ *ltl-next-free* φ ∧ *ltl-next-free* ψ
| *ltl-next-free* (*LTLNext* -) ⟷ *False*
| *ltl-next-free* (*LTLUntil* φ ψ) ⟷ *ltl-next-free* φ ∧ *ltl-next-free* ψ
| *ltl-next-free* (*LTLRelease* φ ψ) ⟷ *ltl-next-free* φ ∧ *ltl-next-free* ψ

Conversion between the two LTL formalizations

primrec *cnv* :: *'a LTL.ltl* ⇒ *'a set PLTL.pltl* **where**
cnv LTLTrue = *PLTL.true*
| *cnv LTLFalse* = *PLTL.false*
| *cnv* (*LTLProp a*) = *PLTL.atom* (*op* ∈ *a*)
| *cnv* (*LTLNeg* φ) = *PLTL.not* (*cnv* φ)
| *cnv* (*LTLAnd* φ ψ) = *PLTL.and* (*cnv* φ) (*cnv* ψ)
| *cnv* (*LTLOr* φ ψ) = *PLTL.or* (*cnv* φ) (*cnv* ψ)

| *cnv* (*LTLNext* φ) = *PLTL.next* (*cnv* φ)
| *cnv* (*LTLUntil* φ ψ) = *PLTL.until* (*cnv* φ) (*cnv* ψ)
| *cnv* (*LTLRelease* φ ψ)
= *PLTL.not* (*PLTL.until* (*PLTL.not* (*cnv* φ)) (*PLTL.not* (*cnv* ψ)))

lemma *PLTL-holds-of-cnv*[*simp*]: *PLTL.holds-of* r (*cnv* φ) $\longleftrightarrow$ *ltl-semantics* r φ
by (*induction* φ *arbitrary*: r) *simp-all*

lemma *PLTL-next-free-cnv*[*simp*]: *PLTL.next-free* (*cnv* φ) $\longleftrightarrow$ *ltl-next-free* φ
by (*induction* φ) *auto*

theorem *next-free-stutter-invariant*:
assumes *next-free*: *ltl-next-free* φ
assumes *eq*: $r \approx r'$
shows $r \models \varphi \longleftrightarrow r' \models \varphi$
— A next free formula cannot distinguish between stutter-equivalent runs.
proof −
{
fix r r'
assume *eq*: $r \approx r'$ **and** *holds*: $r \models \varphi$
hence *holds-of* r (*cnv* φ) **by** *simp*

from *next-free-stutter-invariant*[*of cnv* φ] *next-free*
have *PLTL.stutter-invariant* (*cnv* φ) **by** *simp*
from *stutter-invariantD*[*OF this eq*] *holds* **have** $r' \models \varphi$ **by** *simp*
} **note** *aux*=*this*

from *aux*[*of r r'*] *aux*[*of r' r*] *eq stutter-equiv-sym*[*OF eq*] **show** *?thesis*
by *blast*
qed

context begin interpretation *LTL-Syntax* **.**
primrec *ltlc-next-free* :: *'a ltlc* ⇒ *bool*
where
ltlc-next-free $true_c$ = *True*
| *ltlc-next-free* $false_c$ = *True*
| *ltlc-next-free* ($prop_c(q)$) = *True*
| *ltlc-next-free* (not_c φ) = *ltlc-next-free* φ
| *ltlc-next-free* (φ and_c ψ) = (*ltlc-next-free* φ ∧ *ltlc-next-free* ψ)
| *ltlc-next-free* (φ or_c ψ) = (*ltlc-next-free* φ ∧ *ltlc-next-free* ψ)
| *ltlc-next-free* (φ $implies_c$ ψ) = (*ltlc-next-free* φ ∧ *ltlc-next-free* ψ)
| *ltlc-next-free* (φ iff_c ψ) = (*ltlc-next-free* φ ∧ *ltlc-next-free* ψ)
| *ltlc-next-free* (X_c φ) = *False*
| *ltlc-next-free* (F_c φ) = *ltlc-next-free* φ
| *ltlc-next-free* (G_c φ) = *ltlc-next-free* φ
| *ltlc-next-free* (φ U_c ψ) = (*ltlc-next-free* φ ∧ *ltlc-next-free* ψ)

```
  | ltlc-next-free (φ V_c ψ) = (ltlc-next-free φ ∧ ltlc-next-free ψ)
end

lemma ltlc-to-ltl-next-free-iff:
  ltl-next-free (ltlc-to-ltl φ) ⟷ ltlc-next-free φ
  by (induction φ) (auto simp: Let-def)


theorem ltlc-next-free-stutter-invariant:
  assumes next-free: ltlc-next-free φ
  assumes eq: r ≈ r'
  shows r ⊨_c φ ⟷ r' ⊨_c φ
  — A next free formula cannot distinguish between stutter-equivalent runs.
proof −
  note NF' = next-free[folded ltlc-to-ltl-next-free-iff]
  from next-free-stutter-invariant[OF NF' eq] show ?thesis
    by (auto simp: ltlc-to-ltl-equiv)
qed

end
```

5 LTL to GBA translation

```
theory LTL-to-GBA
imports
  ../CAVA-Automata/CAVA-Base/CAVA-Base
  LTL LTL-Rewrite
  ../CAVA-Automata/Automata
begin
```

5.1 Statistics

```
code-printing
  code-module Gerth-Statistics ⇀ (SML) ⟪
    structure Gerth-Statistics = struct
      val active = Unsynchronized.ref false
      val data = Unsynchronized.ref (0,0,0)

      fun is-active () = !active
      fun set-data num-states num-init num-acc = (
        active := true;
        data := (num-states, num-init, num-acc)
      )

      fun to-string () = let
        val (num-states, num-init, num-acc) = !data
      in
          Num states:  ^ IntInf.toString (num-states) ^ \n
        ^   Num initial:  ^ IntInf.toString num-init ^ \n
```

```
      ^ Num acc−classes: ^ IntInf.toString num-acc ^ \n
    end

    val - = Statistics.register-stat (Gerth LTL-to-GBA,is-active,to-string)
  end
⟩⟩
code-reserved SML Gerth-Statistics

consts
  stat-set-data-int :: integer ⇒ integer ⇒ integer ⇒ unit

code-printing
  constant stat-set-data-int ⇀ (SML) Gerth'-Statistics.set'-data

definition stat-set-data ns ni na
  ≡ stat-set-data-int (integer-of-nat ns) (integer-of-nat ni) (integer-of-nat na)

lemma [autoref-rules]:
  (stat-set-data,stat-set-data) ∈ nat-rel → nat-rel → nat-rel → unit-rel
  by auto

abbreviation stat-set-data-nres ns ni na ≡ RETURN (stat-set-data ns ni na)

lemma discard-stat-refine[refine]:
  m1≤m2 ⟹ stat-set-data-nres ns ni na ≫ m1 ≤ m2 by simp-all
```

5.2 Preliminaries

Some very special lemmas for reasoning about the nres-monad

```
lemma SPEC-rule-nested2:
  ⟦m ≤ SPEC P; ⋀r1 r2. P (r1, r2) ⟹ g (r1, r2) ≤ SPEC P⟧
  ⟹ m ≤ SPEC (λr'. g r' ≤ SPEC P)
  by (simp add: pw-le-iff refine-pw-simps) blast

lemma SPEC-rule-param2:
  assumes f x ≤ SPEC (P x)
      and ⋀r1 r2. (P x) (r1, r2) ⟹ (P x') (r1, r2)
    shows f x ≤ SPEC (P x')
  using assms
  by (simp add: pw-le-iff refine-pw-simps)

lemma SPEC-rule-weak:
  assumes f x ≤ SPEC (Q x) and f x ≤ SPEC (P x)
      and ⋀r1 r2. ⟦(Q x) (r1, r2); (P x) (r1, r2)⟧ ⟹ (P x') (r1, r2)
    shows f x ≤ SPEC (P x')
  using assms
  by (simp add: pw-le-iff refine-pw-simps)

lemma SPEC-rule-weak-nested2: ⟦f ≤ SPEC Q; f ≤ SPEC P;
```

$\bigwedge r1\ r2.\ [\![Q\ (r1,\ r2);\ P\ (r1,\ r2)]\!] \Longrightarrow g\ (r1,\ r2) \leq SPEC\ P]\!]$
$\Longrightarrow f \leq SPEC\ (\lambda r'.\ g\ r' \leq SPEC\ P)$
by (*simp add*: *pw-le-iff refine-pw-simps*) *blast*

5.3 Creation of States

In this section, the first part of the algorithm, which creates the states of the automaton, is formalized.

type-synonym *node-name* = *nat*

type-synonym
'a frml = *'a ltln*

type-synonym
'a interprt = *'a set word*

record *'a node* =
name :: *node-name*
incoming :: *node-name set*
new :: *'a frml set*
old :: *'a frml set*
next :: *'a frml set*

context begin interpretation *LTL-Syntax* **.**

fun *frml-neg*
where
frml-neg $true_n$ = $false_n$
| *frml-neg* $false_n$ = $true_n$
| *frml-neg* $prop_n(q)$ = $nprop_n(q)$
| *frml-neg* $nprop_n(q)$ = $prop_n(q)$

fun *new1* **where**
new1 $(\mu\ and_n\ \psi) = \{\mu,\psi\}$
| *new1* $(\mu\ U_n\ \psi) = \{\mu\}$
| *new1* $(\mu\ V_n\ \psi) = \{\psi\}$
| *new1* $(\mu\ or_n\ \psi) = \{\mu\}$
| *new1* - = {}

fun *next1* **where**
next1 $(X_n\ \psi) = \{\psi\}$
| *next1* $(\mu\ U_n\ \psi) = \{\mu\ U_n\ \psi\}$
| *next1* $(\mu\ V_n\ \psi) = \{\mu\ V_n\ \psi\}$
| *next1* - = {}

fun *new2* **where**
new2 $(\mu\ U_n\ \psi) = \{\psi\}$
| *new2* $(\mu\ V_n\ \psi) = \{\mu, \psi\}$
| *new2* $(\mu\ or_n\ \psi) = \{\psi\}$

```
| new2 - = {}



definition expand-init ≡ 0
definition expand-new-name ≡ Suc

lemma expand-new-name-expand-init:
  shows ∀ nm. expand-init < expand-new-name nm
by (auto simp add:expand-init-def expand-new-name-def)

lemma expand-new-name-step[intro]:
  shows ⋀n. name (n::'a node) < expand-new-name (name n)
proof −
  fix n::'a node
 show name n < expand-new-name (name n) by (auto simp add:expand-new-name-def)
qed

lemma expand-new-name--less-zero[intro]: 0 < expand-new-name nm
proof −
   from expand-new-name-expand-init have expand-init < expand-new-name nm
by auto
  thus ?thesis by (metis gr0I less-zeroE)
qed

abbreviation
  upd-incoming-f n ≡ (λn'.
    if (old n' = old n ∧ next n' = next n) then
      n'(| incoming := incoming n ∪ incoming n' |)
    else n'
  )

definition
 upd-incoming n ns ≡ ((upd-incoming-f n) ‘ ns)

lemma upd-incoming--elem:
  assumes nd∈upd-incoming n ns
    shows nd∈ns
          ∨ (∃ nd'∈ns. nd = nd'(| incoming := incoming n ∪ incoming nd' |) ∧
                       old nd' = old n ∧
                       next nd' = next n)
proof −
  obtain nd' where nd'∈ns and nd-eq: nd = upd-incoming-f n nd'
    using assms unfolding upd-incoming-def by blast
  thus ?thesis by auto
qed

lemma upd-incoming--ident-node:
```

assumes *nd*∈*upd-incoming n ns* **and** *nd*∈*ns*
shows *incoming n* ⊆ *incoming nd* ∨ ¬ (*old nd* = *old n* ∧ *next nd* = *next n*)
proof(*rule ccontr*)
assume ¬ *?thesis*
hence *nsubset*: ¬ (*incoming n* ⊆ *incoming nd*) **and**
old-next-eq: *old nd* = *old n* ∧ *next nd* = *next n*
by *auto*
moreover
obtain *nd′* **where** *nd′*∈*ns* **and** *nd-eq*: *nd* = *upd-incoming-f n nd′*
using *assms* **unfolding** *upd-incoming-def* **by** *blast*
{ assume *old nd′* = *old n* ∧ *next nd′* = *next n*
with *nd-eq* **have** *nd* = *nd′*(|*incoming* := *incoming n* ∪ *incoming nd′*|) **by** *auto*
with *nsubset* **have** *False* **by** *auto* **}**
moreover
{ assume ¬ (*old nd′* = *old n* ∧ *next nd′* = *next n*)
with *nd-eq old-next-eq* **have** *False* **by** *auto* **}**
ultimately show *False* **by** *fast*
qed

lemma *upd-incoming--ident*:
assumes ∀ *n*∈*ns*. *P n*
and ⋀*n*. ⟦*n*∈*ns*; *P n*⟧ ⟹ (⋀*v*. *P* (*n*(| *incoming* := *v* |)))
shows ∀ *n*∈*upd-incoming n ns*. *P n*
proof
fix *nd v*
let *?f* = λ*n′*.
if (*old n′* = *old n* ∧ *next n′* = *next n*) *then*
n′(| *incoming* := *incoming n* ∪ *incoming n′* |)
else n′
assume *nd*∈*upd-incoming n ns*
then obtain *nd′* **where** *nd′*∈*ns* **and** *nd-eq*: *nd* = *?f nd′*
unfolding *upd-incoming-def* **by** *blast*
{ assume *old nd′* = *old n* ∧ *next nd′* = *next n*
then obtain *v* **where** *nd* = *nd′*(| *incoming* := *v* |) **using** *nd-eq* **by** *auto*
with *assms* ‹*nd′*∈*ns*› **have** *P nd* **by** *auto* **}**
thus *P nd* **using** *nd-eq* ‹*nd′*∈*ns*› *assms* **by** *auto*
qed

lemma *name-upd-incoming-f*[*simp*]: *name* (*upd-incoming-f n x*) = *name x*
by *auto*

lemma *name-upd-incoming*[*simp*]:
name ‘ (*upd-incoming n ns*) = *name* ‘ *ns* (**is** *?lhs* = *?rhs*)
unfolding *upd-incoming-def*
proof *safe*
fix *x*
assume *x*∈*ns*

hence *upd-incoming-f n x* ∈ (λ*n'*. *local.upd-incoming-f n n'*) ‘ *ns* **by** *auto*
hence *name* (*upd-incoming-f n x*) ∈ *name* ‘ (λ*n'*. *local.upd-incoming-f n n'*) ‘ *ns*
by *blast*
thus *name x* ∈ *name* ‘ (λ*n'*. *local.upd-incoming-f n n'*) ‘ *ns* **by** *simp*
qed *auto*

abbreviation *expand-body*
where
expand-body ≡ (λ*expand* (*n*,*ns*).
if new n = {} *then* (
if (∃ *n'*∈*ns*. *old n'* = *old n* ∧ *next n'* = *next n*) *then*
RETURN (*name n*, *upd-incoming n ns*)
else
expand (
(|
name=*expand-new-name* (*name n*),
incoming={*name n*},
new=*next n*,
old={},
next={}
|),
{*n*} ∪ *ns*)
) *else do* {
φ ← *SPEC* (λ*x*. *x*∈(*new n*));
let n = *n*(| *new* := *new n* − {φ} |);
if (∃ *q*. φ = *prop*$_n$(*q*) ∨ φ = *nprop*$_n$(*q*)) *then*
(*if* (*frml-neg* φ)∈*old n then RETURN* (*name n*, *ns*)
else expand (*n*(| *old* := {φ} ∪ *old n* |), *ns*))
else if φ = *true*$_n$ *then expand* (*n*(| *old* := {φ} ∪ *old n* |), *ns*)
else if φ = *false*$_n$ *then RETURN* (*name n*, *ns*)
else if (∃ ν μ. (φ = ν *and*$_n$ μ) ∨ (φ = *X*$_n$ ν)) *then*
expand (
n(|
new := *new1* φ ∪ *new n*,
old := {φ} ∪ *old n*,
next := *next1* φ ∪ *next n*
|),
ns)
else do {
(*nm*, *nds*) ← *expand* (
n(|
new := *new1* φ ∪ *new n*,
old := {φ} ∪ *old n*,
next := *next1* φ ∪ *next n*
|),
ns);
expand (*n*(| *name* := *nm*, *new* := *new2* φ ∪ *new n*, *old* := {φ} ∪ *old n* |),
nds)

}
}
)

lemma *expand-body-mono*: *trimono expand-body* **by** *refine-mono*

definition *expand* :: (*'a node* × (*'a node set*)) ⇒ (*node-name* × *'a node set*) *nres*
where *expand* ≡ *REC expand-body*

lemma *REC-rule-old*:
fixes *x*::*'x*
assumes *M*: *trimono body*
assumes *I0*: Φ *x*
assumes *IS*: ⋀*f x*. ⟦ ⋀*x*. Φ *x* ⟹ *f x* ≤ *M x*; Φ *x*; *f* ≤ *REC body* ⟧
⟹ *body f x* ≤ *M x*
shows *REC body x* ≤ *M x*
by (*rule REC-rule-arb*[**where** *pre*=λ-. Φ **and** *M*=λ-. *M*, *OF assms*])

lemma *expand-rec-rule*:
assumes *I0*: Φ *x*
assumes *IS*: ⋀*f x*. ⟦ ⋀*x*. *f x* ≤ *expand x*; ⋀*x*. Φ *x* ⟹ *f x* ≤ *M x*; Φ *x* ⟧
⟹ *expand-body f x* ≤ *M x*
shows *expand x* ≤ *M x*
unfolding *expand-def*
apply (*rule REC-rule-old*[**where** Φ=Φ])
apply (*rule expand-body-mono*)
apply (*rule I0*)
apply (*rule IS*[*unfolded expand-def*])
apply (*blast dest*: *le-funD*)
apply *blast*
apply *blast*
done

abbreviation
expand-assm-incoming n-ns
≡ (∀ *nm*∈*incoming* (*fst n-ns*). *name* (*fst n-ns*) > *nm*)
∧ *0* < *name* (*fst n-ns*)
∧ (∀ *q*∈*snd n-ns*.
name (*fst n-ns*) > *name q*
∧ (∀ *nm*∈*incoming q*. *name* (*fst n-ns*) > *nm*))

abbreviation
expand-rslt-incoming nm-nds
≡ (∀ *q*∈*snd nm-nds*. (*fst nm-nds* > *name q* ∧ (∀ *nm'*∈*incoming q*. *fst nm-nds* > *nm'*)))

abbreviation
expand-rslt-name n-ns nm-nds
≡ (*name* (*fst n-ns*) ≤ *fst nm-nds* ∧ *name* ‘ (*snd n-ns*) ⊆ *name* ‘ (*snd nm-nds*))

$\wedge$ *name* ‘ (*snd nm-nds*)
= *name* ‘ (*snd n-ns*) $\cup$ *name* ‘ {*nd*$\in$*snd nm-nds*. *name nd* $\geq$ *name* (*fst n-ns*)}

abbreviation
expand-name-ident nds
$\equiv$ ($\forall\, q{\in}nds.\ \exists!\, q'{\in}nds.$ *name* q = *name* q')

abbreviation
expand-assm-exist ξ *n-ns*
$\equiv$ {$\eta.\ \exists\, \mu.\ \mu\ U_n\ \eta \in$ *old* (*fst n-ns*) $\wedge\ \xi \models_n \eta$} $\subseteq$ *new* (*fst n-ns*) $\cup$ *old* (*fst n-ns*)
$\wedge$ ($\forall\, \psi{\in}$*new* (*fst n-ns*). $\xi \models_n \psi$)
$\wedge$ ($\forall\, \psi{\in}$*old* (*fst n-ns*). $\xi \models_n \psi$)
$\wedge$ ($\forall\, \psi{\in}$*next* (*fst n-ns*). $\xi \models_n X_n\ \psi$)

abbreviation
expand-rslt-exist--node-prop ξ *n nd*
$\equiv$ *incoming n* $\subseteq$ *incoming nd*
$\wedge$ ($\forall\, \psi{\in}$*old nd*. $\xi \models_n \psi$) $\wedge$ ($\forall\, \psi{\in}$*next nd*. $\xi \models_n X_n\ \psi$)
$\wedge$ {$\eta.\ \exists\, \mu.\ \mu\ U_n\ \eta \in$ *old nd* $\wedge\ \xi \models_n \eta$} $\subseteq$ *old nd*

abbreviation
expand-rslt-exist ξ *n-ns nm-nds*
$\equiv$ ($\exists\, nd{\in}$*snd nm-nds*. *expand-rslt-exist--node-prop* ξ (*fst n-ns*) *nd*)

abbreviation
expand-rslt-exist-eq--node n nd
$\equiv$ *name n* = *name nd*
$\wedge$ *old n* = *old nd*
$\wedge$ *next n* = *next nd*
$\wedge$ *incoming n* $\subseteq$ *incoming nd*

abbreviation
expand-rslt-exist-eq n-ns nm-nds $\equiv$
($\forall\, n{\in}$*snd n-ns*. $\exists\, nd{\in}$*snd nm-nds*. *expand-rslt-exist-eq--node n nd*)

lemma *expand-name-propag*:
assumes *expand-assm-incoming n-ns* $\wedge$ *expand-name-ident* (*snd n-ns*) (**is** *?Q n-ns*)
shows *expand n-ns* $\leq$ *SPEC* ($\lambda r.$ *expand-rslt-incoming r*
$\wedge$ *expand-rslt-name n-ns r*
$\wedge$ *expand-name-ident* (*snd r*))
(**is** *expand* - $\leq$ *SPEC* (*?P n-ns*))
using *assms*
proof (*rule-tac expand-rec-rule*[**where** Φ=*?Q*], *simp*, *intro refine-vcg*)
case (*goal1* - - *n ns*)
hence *Q*: *?Q* (*n*, *ns*) **by** *fast*
let *?nds* = *upd-incoming n ns*

```
have ∀ q∈?nds. name q < name n using goal1
  by (rule-tac upd-incoming--ident) auto
moreover
have ∀ q∈?nds. ∀ nm'∈incoming q. nm' < name n (is ∀ q∈-. ?sg q)
proof
  fix q
  assume q-in:q∈?nds
  { assume q∈ns
    hence ?sg q using goal1 by auto }
  moreover
  { assume q∉ns
    with upd-incoming--elem[OF q-in]
    obtain nd' where
      nd'-def:nd'∈ns ∧ q = nd'(|incoming := incoming n ∪ incoming nd'|)
    by blast

    { fix nm'
      assume nm'∈incoming q
      moreover
      { assume nm'∈incoming n
        with Q have nm' < name n by auto }
      moreover
      { assume nm'∈incoming nd'
        with Q nd'-def have nm' < name n by auto }
      ultimately have nm' < name n using nd'-def by auto }

    hence ?sg q by fast }
  ultimately show ?sg q by fast
qed
moreover
have expand-name-ident ?nds
proof (rule-tac upd-incoming--ident)
  case goal1 show ?case
    proof
      fix q
      assume q∈ns

      with Q have ∃!q'∈ns. name q = name q' by auto
      then obtain q' where q'∈ns and name q = name q'
                       and q'-all: ∀ q''∈ns. name q' = name q'' ⟶ q' = q''
        by auto
      let ?q' = upd-incoming-f n q'
      have P-a: ?q'∈?nds ∧ name q = name ?q'
        using ‹q'∈ns› ‹name q = name q'› q'-all
        unfolding upd-incoming-def by auto

      have P-all: ∀ q''∈?nds. name ?q' = name q'' ⟶ ?q' = q''
      proof(clarify)
        fix q''
```

```
        assume q''∈?nds and q''-name-eq: name ?q' = name q''
        { assume q''∉ns
          with upd-incoming--elem[OF ‹q''∈?nds›]
          obtain nd'' where
            nd''∈ns
            and q''-is: q'' = nd''(|incoming := incoming n ∪ incoming nd''|)
                        ∧ old nd'' = old n ∧ next nd'' = next n
            by auto
          hence name nd'' = name q'
            using q''-name-eq
            by (cases old q' = old n ∧ next q' = next n) auto
          with ‹nd''∈ns› q'-all have nd'' = q' by auto
          hence ?q' = q'' using q''-is by simp }
        moreover
        { assume q''∈ns
          moreover
          have name q' = name q''
            using q''-name-eq
            by (cases old q' = old n ∧ next q' = next n) auto
          moreover
          hence incoming n ⊆ incoming q''
            ⟹ incoming q'' = incoming n ∪ incoming q''
            by auto
          ultimately have ?q' = q''
            using upd-incoming--ident-node[OF ‹q''∈?nds›] q'-all
            by auto
        }
        ultimately show ?q' = q'' by fast
      qed

      show ∃!q'∈upd-incoming n ns. name q = name q'
      proof(rule-tac ex1I[of - ?q'])
        case goal1 thus ?case using P-a by simp
      next
        case goal2 thus ?case
          using P-all unfolding P-a[THEN conjunct2, THEN sym]
          by blast
      qed
    qed
  qed simp

  ultimately show ?case using goal1 by auto
next
  case (goal2 f - n ns)
  hence step: ⋀x. ?Q x ⟹ f x ≤ SPEC (?P x) by simp
  have Q: ?Q (n, ns) using goal2 by auto

  show ?case unfolding ‹x = (n, ns)›
  proof (rule-tac SPEC-rule-param2[where P = ?P], rule-tac step)
```

```
      case goal1
        with expand-new-name-step[of n] show ?case
          using Q
          apply (auto elim: order-less-trans[rotated])
          done
    next
      case (goal2 - nds)
        hence name ‘ ns ⊆ name ‘ nds using goal2 by auto
        with expand-new-name-step[of n] show ?case using goal2
          by auto
    qed
next
  case goal3 thus ?case using goal3 by auto
next
  case (goal4 f)
    hence step: ⋀x. ?Q x ⟹ f x ≤ SPEC (?P x) by simp+
    show ?case using goal4 by (rule-tac SPEC-rule-param2, rule-tac step) auto
next
  case (goal5 f)
    hence step: ⋀x. ?Q x ⟹ f x ≤ SPEC (?P x) by simp+
    show ?case using goal5 by (rule-tac SPEC-rule-param2, rule-tac step) auto
next
  case goal6 thus ?case by auto
next
  case (goal7 f)
    hence step: ⋀x. ?Q x ⟹ f x ≤ SPEC (?P x) by simp+
    show ?case using goal7 by (rule-tac SPEC-rule-param2, rule-tac step) auto
next
  case (goal8 f x n ns ψ)
    hence goal-assms: ψ ∈ new n ∧ (∃ν μ. ψ = ν or_n μ ∨ ψ = ν U_n μ ∨ ψ = ν
V_n μ)
      by (cases ψ) auto
    have Q: ?Q (n, ns) and step: ⋀x. ?Q x ⟹ f x ≤ SPEC (?P x)
      using goal8 by simp+
    show ?case using goal-assms Q unfolding case-prod-unfold ‹x = (n, ns)›
    proof (rule-tac SPEC-rule-nested2)
      case goal1 thus ?case
        by (rule-tac SPEC-rule-param2, rule-tac step) auto
    next
      case (goal2 nm nds)
        hence P-x: (name n ≤ nm ∧ name ‘ ns ⊆ name ‘ nds)
                  ∧ name ‘ nds = name ‘ ns ∪ name ‘ {nd∈nds. name nd ≥ name
n}
              (is - ∧ ?nodes-eq nds ns (name n)) by auto

        show ?case using goal2
        proof (rule-tac SPEC-rule-param2[where P = ?P])
          case goal1 thus ?case by (rule-tac step) auto
        next
```

```
        case (goal2 nm' nds')
         hence ∀ q∈nds'. name q < nm' ∧ (∀ nm''∈incoming q. nm'' < nm') by
auto
          moreover
          have ?nodes-eq nds ns (name n) and ?nodes-eq nds' nds nm
           and name n ≤ nm using goal2 P-x by auto
          hence ?nodes-eq nds' ns (name n) by auto
          hence expand-rslt-name (n, ns) (nm', nds')
          using goal2 P-x subset-trans[of name ` ns name ` nds] by auto
          ultimately show ?case using goal2 by auto
       qed
   qed
qed

lemmas expand-name-propag--incoming = SPEC-rule-conjunct1[OF expand-name-propag]
lemmas expand-name-propag--name =
  SPEC-rule-conjunct1[OF SPEC-rule-conjunct2[OF expand-name-propag]]
lemmas expand-name-propag--name-ident =
  SPEC-rule-conjunct2[OF SPEC-rule-conjunct2[OF expand-name-propag]]


lemma expand-rslt-exist-eq:
   shows expand n-ns ≤ SPEC (expand-rslt-exist-eq n-ns)
     (is - ≤ SPEC (?P n-ns))
using assms
proof (rule-tac expand-rec-rule[where Φ=λ-. True], simp, intro refine-vcg)
  case (goal1 f - n ns)
   let ?r = (name n, upd-incoming n ns)
   have expand-rslt-exist-eq (n, ns) ?r unfolding snd-conv
   proof
     fix n'
     assume n'∈ns
     { assume old n' = old n ∧ next n' = next n
       with ‹n'∈ns›
       have n'(| incoming := incoming n ∪ incoming n' |) ∈ upd-incoming n ns
         unfolding upd-incoming-def by auto
     }
     moreover
     { assume ¬ (old n' = old n ∧ next n' = next n)
       with ‹n'∈ns› have n' ∈ upd-incoming n ns
         unfolding upd-incoming-def by auto
     }
     ultimately show ∃ nd∈upd-incoming n ns. expand-rslt-exist-eq--node n' nd
       by force
   qed
   thus ?case using goal1 by auto
next
  case (goal2 f)
   hence step: ⋀x. f x ≤ SPEC (?P x) by simp
```

```
    show ?case using goal2 by (rule-tac SPEC-rule-param2, rule-tac step) auto
next
  case goal3 thus ?case by auto
next
  case (goal4 f)
    hence step: ⋀x. f x ≤ SPEC (?P x) by simp
    show ?case using goal4 by (rule-tac SPEC-rule-param2, rule-tac step) auto
next
  case (goal5 f)
    hence step: ⋀x. f x ≤ SPEC (?P x) by simp
    show ?case using goal5 by (rule-tac SPEC-rule-param2, rule-tac step) auto
next
  case goal6 thus ?case by auto
next
  case (goal7 f)
    hence step: ⋀x. f x ≤ SPEC (?P x) by simp
    show ?case using goal7 by (rule-tac SPEC-rule-param2, rule-tac step) auto
next
  case (goal8 f x n ns)
  have step: ⋀x. f x ≤ SPEC (?P x) using goal8 by simp

  show ?case using goal8
  proof (rule-tac SPEC-rule-nested2)
    case goal1 thus ?case by (rule-tac SPEC-rule-param2, rule-tac step) auto
  next
    case (goal2 nm nds)
    have P-x: ?P (n, ns) (nm, nds) using goal2 by fast

    show ?case unfolding case-prod-unfold ‹x = (n, ns)›
    proof (rule-tac SPEC-rule-param2[where P = ?P])
      case goal1 thus ?case by (rule-tac step)
    next
      case (goal2 nm' nds')
      { fix n'
        assume n'∈ns
        with P-x obtain nd where
          nd∈nds and
          n'-split: expand-rslt-exist-eq--node n' nd
          by auto

        with goal2 obtain nd' where
          nd'∈nds'
          and expand-rslt-exist-eq--node nd nd'
          by auto
        hence ∃nd'∈nds'. expand-rslt-exist-eq--node n' nd'
          using n'-split subset-trans[of incoming n'] by auto
      }
      hence expand-rslt-exist-eq (n, ns) (nm', nds') by auto
      thus ?case using goal2 by auto
```

```
    qed
  qed
qed

lemma expand-prop-exist:
  shows expand n-ns
  ≤ SPEC (λr. expand-assm-exist ξ n-ns ⟶ expand-rslt-exist ξ n-ns r)
  (is - ≤ SPEC (?P n-ns))
  using assms
proof (rule-tac expand-rec-rule[where Φ=λ-. True], simp, intro refine-vcg)
  case (goal1 f - n ns)
    let ?nds = upd-incoming n ns
    let ?r = (name n, ?nds)
    { assume Q: expand-assm-exist ξ (n, ns)
      note ‹∃ n'∈ns. old n' = old n ∧ next n' = next n›
      then obtain n' where n'∈ns and assm-eq: old n' = old n ∧ next n' = next
n
        by auto
      let ?nd = n'(| incoming := incoming n ∪ incoming n' |)
       have ?nd ∈ ?nds using ‹n'∈ns› assm-eq unfolding upd-incoming-def by
auto
      moreover
      have incoming n ⊆ incoming ?nd by auto
      moreover
      have expand-rslt-exist--node-prop ξ n ?nd using Q assm-eq ‹new n = {}›
        by simp
      ultimately have expand-rslt-exist ξ (n, ns) ?r
        unfolding fst-conv snd-conv by blast
    }
    thus ?case using goal1 by auto
next
  case (goal2 f x n ns)
    hence step: ⋀x. f x ≤ SPEC (?P x)
      and f-sup: ⋀x. f x ≤ expand x by auto
    show ?case unfolding ‹x = (n, ns)›
    proof (rule-tac SPEC-rule-weak[where Q = expand-rslt-exist-eq])
      case goal1 thus ?case
        by (rule-tac order-trans, rule-tac f-sup, rule-tac expand-rslt-exist-eq)
    next
      case goal2 thus ?case by (rule-tac step)
    next
      case (goal3 nm nds)
        hence name ' ns ⊆ name ' nds using goal3 by auto
        moreover
        { assume assm-ex: expand-assm-exist ξ (n, ns)
          then obtain nd where nd∈nds and expand-rslt-exist-eq--node n nd
          using goal3 by force+
          hence expand-rslt-exist--node-prop ξ n nd
           using assm-ex ‹new n = {}› by auto
```

```
      hence expand-rslt-exist ξ (n, ns) (nm, nds) using ‹nd∈nds› by auto }
    ultimately show ?case using expand-new-name-step[of n] goal3 by auto
  qed
next
  case (goal3 f x n ns ψ)
  { assume expand-assm-exist ξ (n, ns)
    hence ξ ⊨_n ψ and ξ ⊨_n frml-neg ψ using goal3 by auto
    hence False using goal3 by auto }
  thus ?case using goal3 by auto
next
  case (goal4 f x n ns ψ)
  hence goal-assms: ψ ∈ new n ∧ (∃ q. ψ = prop_n(q) ∨ ψ = nprop_n(q))
    and step: ⋀x. f x ≤ SPEC (?P x) by simp+
  show ?case using goal-assms unfolding ‹x = (n, ns)›
  proof (rule-tac SPEC-rule-param2, rule-tac step)
    case (goal1 nm nds)
    { assume expand-assm-exist ξ (n, ns)
      hence expand-rslt-exist ξ (n, ns) (nm, nds) using goal1 by auto }
    thus ?case by auto
  qed
next
  case (goal5 f x n ns ψ)
  hence goal-assms: ψ ∈ new n ∧ ψ = true_n
    and step: ⋀x. f x ≤ SPEC (?P x) by simp+
  show ?case using goal-assms unfolding ‹x = (n, ns)›
  proof (rule-tac SPEC-rule-param2, rule-tac step)
    case (goal1 nm nds)
    { assume expand-assm-exist ξ (n, ns)
      hence expand-rslt-exist ξ (n, ns) (nm, nds) using goal1 by auto }
    thus ?case by auto
  qed
next
  case (goal6 f x n ns ψ)
  { assume expand-assm-exist ξ (n, ns)
    hence ξ ⊨_n false_n using goal6 by auto }
  thus ?case using goal6 by auto
next
  case (goal7 f x n ns ψ)
  hence goal-assms: ψ ∈ new n ∧ (∃ ν μ. ψ = ν and_n μ ∨ ψ = X_n ν)
    and step: ⋀x. f x ≤ SPEC (?P x) by simp+
  show ?case using goal-assms unfolding ‹x = (n, ns)›
  proof (rule-tac SPEC-rule-param2, rule-tac step)
    case (goal1 nm nds)
    { assume expand-assm-exist ξ (n, ns)
      hence expand-rslt-exist ξ (n, ns) (nm, nds) using goal1 by auto }
    thus ?case by auto
  qed
next
  case (goal8 f x n ns ψ)
```

hence *goal-assms*: $\psi \in new\ n \wedge (\exists \nu\ \mu.\ \psi = \nu\ or_n\ \mu \vee \psi = \nu\ U_n\ \mu \vee \psi = \nu\ V_n\ \mu)$
by (*cases* ψ) *auto*
have *step*: $\bigwedge x.\ f\ x \leq SPEC\ (?P\ x)$
and *f-sup*: $\bigwedge x.\ f\ x \leq expand\ x$ **using** *goal8* **by** *auto*
let *?x1* = $(n(\!|new := new\ n - \{\psi\},\ new := new1\ \psi \cup new\ (n(\!|new := new\ n - \{\psi\}|\!)),$
$old := \{\psi\} \cup old\ n,\ next := next1\ \psi \cup next\ n|\!),\ ns)$

let *?new1-assm-sel* = $\lambda\psi.$ (*case* ψ *of* $\mu\ U_n\ \eta \Rightarrow \eta \mid \mu\ V_n\ \eta \Rightarrow \mu \mid \mu\ or_n\ \eta \Rightarrow \eta$)

{ assume *new1-assm*: $\neg\ (\xi \models_n (?new1\text{-}assm\text{-}sel\ \psi))$
hence *?case* **using** *goal-assms* **unfolding** ‹$x = (n,\ ns)$›
proof (*rule-tac SPEC-rule-nested2*)
case *goal1* **thus** *?case*
proof (*rule-tac SPEC-rule-param2*, *rule-tac step*)
case (*goal1 nm nds*)
{ assume *expand-assm-exist* ξ (n, ns)
with *goal1* **have** *expand-assm-exist* ξ *?x1* **unfolding** *fst-conv*
proof (*cases* ψ)
case (*goal8* μ η)
hence $\xi \models_n \mu\ U_n\ \eta$ **by** *fast*
hence $\xi \models_n \mu$ **and** $\xi \models_n X_n\ (\mu\ U_n\ \eta)$
using *goal8 ltln-expand-Until*[*of* ξ μ η] **by** *auto*
thus *?case* **using** *goal8* **by** *auto*
next
case (*goal9* μ η)
hence $\xi \models_n \mu\ V_n\ \eta$ **by** *fast*
moreover hence $\xi \models_n \eta$ **and** $\xi \models_n X_n\ (\mu\ V_n\ \eta)$
using *goal9 ltln-expand-Release*[*of* ξ μ η] **by** *auto*
ultimately show *?case* **using** *goal9* **by** *auto*
qed *auto*
hence *expand-rslt-exist* ξ (n, ns) (nm, nds) **using** *goal1* **by** *force* **}**
thus *?case* **using** *goal1* **by** *auto*
qed
next
case (*goal2 nm nds*)
hence *P-x*: *?P* (n, ns) (nm, nds) **by** *fast*

show *?case* **unfolding** *case-prod-unfold*
proof (*rule-tac*
SPEC-rule-weak[**where** P = *?P* **and** Q = *expand-rslt-exist-eq*])
case *goal1* **thus** *?case*
by (*rule-tac order-trans*,
rule-tac f-sup,
rule-tac expand-rslt-exist-eq)
next
case *goal2* **thus** *?case* **by** (*rule-tac step*)

```
    next
      case (goal3 nm' nds')
        { assume expand-assm-exist ξ (n, ns)
          with P-x have expand-rslt-exist ξ (n, ns) (nm, nds) by simp
          then obtain nd where
            nd∈nds and expand-rslt-exist--node-prop ξ n nd
          using goal-assms by auto
          moreover
          with goal3 obtain nd' where
            nd'∈nds' and expand-rslt-exist-eq--node nd nd'
            by force
          ultimately have expand-rslt-exist--node-prop ξ n nd'
          using subset-trans[of incoming n incoming nd] by auto
          hence expand-rslt-exist ξ (n,ns) (nm', nds')
            using ‹nd'∈nds'› goal-assms by auto }
        thus ?case by fast
    qed
  qed }
moreover
{ assume new1-assm: ξ ⊨n (?new1-assm-sel ψ)
  let ?x2f = λ(nm::node-name, nds::'a node set). (
      n(|new := new n − {ψ},
        name := nm,
        new := new2 ψ ∪ new (n(|new := new n − {ψ}|)),
        old := {ψ} ∪ old n|),
      nds)
  have P-x: f ?x1 ≤ SPEC (?P ?x1) by (rule step)
  moreover
  { fix r :: node-name × 'a node set
    let ?x2 = ?x2f r

    assume assm: (?P ?x1) r
    have  f ?x2 ≤ SPEC (?P (n, ns)) unfolding case-prod-unfold fst-conv
    proof(rule-tac SPEC-rule-param2, rule-tac step)
      case (goal1 nm' nds')
        { assume expand-assm-exist ξ (n, ns)
          with new1-assm goal-assms have expand-assm-exist ξ ?x2
          proof (cases r, cases ψ)
            case (goal9 - - μ η)
              hence ξ ⊨n μ V_n η unfolding fst-conv by fast
             moreover with ltln-expand-Release[of ξ μ η] have ξ ⊨n η by auto
              ultimately show ?case using goal9 by auto
          qed auto
          with goal1 have expand-rslt-exist ξ ?x2 (nm', nds')
            unfolding case-prod-unfold fst-conv snd-conv by fast
          hence expand-rslt-exist ξ (n, ns) (nm', nds') by (cases r, auto) }
        thus ?case by simp
    qed }
  hence SPEC (?P ?x1)
```

$\leq$ *SPEC* (λr. (*case r of* (*nm*, *nds*) =>
f (*?x2f* (*nm*, *nds*))) $\leq$ *SPEC* (*?P* (*n*, *ns*)))
using *goal-assms* **by** (*rule-tac SPEC-rule*) *force*
finally have *?case* **unfolding** *case-prod-unfold* ‹$x = (n, ns)$› **by** *simp* **}**
ultimately show *?case* **by** *fast*
qed

Termination proof

definition $expand_T$ :: (*'a node* $\times$ (*'a node set*)) $\Rightarrow$ (*node-name* $\times$ *'a node set*) *nres*
where $expand_T$ *n-ns* $\equiv$ REC_T *expand-body n-ns*

abbreviation
old-next-pair n $\equiv$ (*old n*, *next n*)

abbreviation
old-next-limit φ $\equiv$ *Pow* (*subfrmlsn* φ) $\times$ *Pow* (*subfrmlsn* φ)

lemma *old-next-limit-finite*: *finite* (*old-next-limit* φ)
using *subfrmlsn-finite* **by** *auto*

definition
expand-ord φ $\equiv$
inv-image (*finite-psupset* (*old-next-limit* φ) <*lex*> *less-than*)
(λ(*n*, *ns*). (*old-next-pair* ‘ *ns*, *frmlset-sumn* (*new n*)))

lemma *expand-ord-wf*[*simp*]: *wf* (*expand-ord* φ)
using *finite-psupset-wf*[*OF old-next-limit-finite*] **unfolding** *expand-ord-def* **by** *auto*

abbreviation
expand-inv-node φ *n*
$\equiv$ *finite* (*new n*) $\wedge$ *finite* (*old n*) $\wedge$ *finite* (*next n*)
$\wedge$ (*new n*) $\cup$ (*old n*) $\cup$ (*next n*) $\subseteq$ *subfrmlsn* φ

abbreviation
expand-inv-result φ *ns*
$\equiv$ *finite ns* $\wedge$ ($\forall$ *n'*$\in$*ns*. (*new n'*) $\cup$ (*old n'*) $\cup$ (*next n'*) $\subseteq$ *subfrmlsn* φ)

definition
expand-inv φ *n-ns*
$\equiv$ (*case n-ns of* (*n*, *ns*) $\Rightarrow$ *expand-inv-node* φ *n* $\wedge$ *expand-inv-result* φ *ns*)

lemma *new1-less-setsum*: *frmlset-sumn* (*new1* φ) < *frmlset-sumn* {φ}
proof (*cases* φ, *auto*)
fix ν μ
show *frmlset-sumn* {ν,μ} < *Suc* (*size-frmln* ν + *size-frmln* μ)
by (*cases* $\nu = \mu$) *auto*
qed

lemma *new2-less-setsum*: *frmlset-sumn* (*new2* φ) < *frmlset-sumn* {φ}

proof (*cases* φ, *auto*)
fix ν μ
show *frmlset-sumn* $\{\nu,\mu\}$ < *Suc* (*size-frmln* ν + *size-frmln* μ)
by (*cases* $\nu = \mu$) *auto*
qed

lemma *new1-finite*[*intro*]:*finite* (*new1* ψ) **by** (*cases* ψ) *auto*
lemma *new1-subset-frmls*:$\varphi \in$ *new1* $\psi \Longrightarrow \varphi \in$ *subfrmlsn* ψ **by** (*cases* ψ) *auto*

lemma *new2-finite*[*intro*]:*finite* (*new2* ψ) **by** (*cases* ψ) *auto*
lemma *new2-subset-frmls*:$\varphi \in$ *new2* $\psi \Longrightarrow \varphi \in$ *subfrmlsn* ψ **by** (*cases* ψ) *auto*

lemma *next1-finite*[*intro*]:*finite* (*next1* ψ) **by** (*cases* ψ) *auto*
lemma *next1-subset-frmls*:$\varphi \in$ *next1* $\psi \Longrightarrow \varphi \in$ *subfrmlsn* ψ **by** (*cases* ψ) *auto*

lemma *expand-inv-result* φ *ns* $\Longrightarrow$ *old-next-pair* ‘ *ns* $\subseteq$ *old-next-limit* φ
by *auto*

lemma *expand-inv-impl*[*intro*!]:
assumes *expand-inv* φ (*n*, *ns*)
and *newn*:$\psi \in$*new n*
and *old-next-pair* ‘ *ns* $\subseteq$ *old-next-pair* ‘ *ns*′
and *expand-inv-result* φ *ns*′
and (*n*′ = *n*⦇ *new* := *new n* − {ψ}, *old* := {ψ} ∪ *old n* ⦈) ∨
(*n*′ = *n*⦇ *new* := *new1* ψ ∪ (*new n* − {ψ}),
old := {ψ} ∪ *old n*,
next := *next1* ψ ∪ *next n* ⦈) ∨
(*n*′ = *n*⦇ *name* := *nm*,
new := *new2* ψ ∪ (*new n* − {ψ}), *old* := {ψ} ∪ *old n* ⦈)
(**is** *?case1* ∨ *?case2* ∨ *?case3*)
shows ((*n*′, *ns*′), (*n*, *ns*))∈*expand-ord* φ ∧ *expand-inv* φ (*n*′, *ns*′)
(**is** *?concl1* ∧ *?concl2*)
proof
{ assume *n*′*def*:*?case1*
with *assms* **have** *?concl1*
unfolding *expand-inv-def expand-ord-def finite-psupset-def* **by** *auto*
}
moreover
{ assume *n*′*def*:*?case2*
have ψ*innew*:$\psi \in$ *new n* **and** *fin-new*:*finite* (*new n*)
using *assms* **unfolding** *expand-inv-def* **by** *auto*
from ψ*innew setsum-diff1-nat*[*of size-frmln new n* ψ]
have *frmlset-sumn* (*new n* − {ψ}) = *frmlset-sumn* (*new n*) − *size-frmln* ψ
by *simp*
moreover
from *fin-new setsum-Un-nat*[*OF new1-finite* -, *of new n* − {ψ} *size-frmln* ψ]
have *frmlset-sumn* (*new n*′)
≤ *frmlset-sumn* (*new1* ψ) + *frmlset-sumn* (*new n* − {ψ})
unfolding *n*′*def* **by** *auto*

```
    moreover have sum-leq:frmlset-sumn (new n) ≥ frmlset-sumn {ψ}
      using ψinnew setsum-mono2[OF fin-new, of {ψ} size-frmln]
        size-frmln-gt-zero
      by simp
    ultimately have frmlset-sumn (new n′) < frmlset-sumn (new n)
      using new1-less-setsum[of ψ] sum-leq by auto
     hence ?concl1 using assms unfolding expand-ord-def finite-psupset-def by
auto }
  moreover
  { assume n′def:?case3
    have ψinnew:ψ∈ new n and fin-new:finite (new n)
      using assms unfolding expand-inv-def by auto
    from ψinnew setsum-diff1-nat[of size-frmln new n ψ]
    have frmlset-sumn (new n − {ψ}) = frmlset-sumn (new n) − size-frmln ψ
      by simp
    moreover
    from fin-new setsum-Un-nat[OF new2-finite -, of new n − {ψ} size-frmln ψ]
    have frmlset-sumn (new n′)
      ≤ frmlset-sumn (new2 ψ) + frmlset-sumn (new n − {ψ})
      unfolding n′def by auto
    moreover have sum-leq:frmlset-sumn (new n) ≥ frmlset-sumn {ψ}
      using ψinnew setsum-mono2[OF fin-new, of {ψ} size-frmln]
        size-frmln-gt-zero
      by simp
    ultimately have frmlset-sumn (new n′) < frmlset-sumn (new n)
      using new2-less-setsum[of ψ] sum-leq by auto
     hence ?concl1 using assms unfolding expand-ord-def finite-psupset-def by
auto }
  ultimately show ?concl1 using assms by fast
next
  have new1 ψ ⊆ subfrmlsn φ
    and new2 ψ ⊆ subfrmlsn φ
    and next1 ψ ⊆ subfrmlsn φ
  using assms subfrmlsn-subset[OF new1-subset-frmls[of - ψ]]
             subfrmlsn-subset[of ψ φ, OF set-rev-mp[of - new n]]
             subfrmlsn-subset[OF new2-subset-frmls[of - ψ]]
             subfrmlsn-subset[OF next1-subset-frmls[of - ψ]]
   unfolding expand-inv-def

   apply −
   apply (clarsimp split: prod.splits)
   apply (metis in-mono new1-subset-frmls)
   apply (clarsimp split: prod.splits)
   apply (metis new2-subset-frmls set-rev-mp)
   apply (clarsimp split: prod.splits)
   apply (metis in-mono next1-subset-frmls)
   done

  thus ?concl2 using assms
```

```
    unfolding expand-inv-def
    by auto
qed

lemma expand-term-prop-help:
  assumes ((n', ns'), (n, ns))∈expand-ord φ ∧ expand-inv φ (n', ns')
  and assm-rule: ⟦expand-inv φ (n', ns'); ((n', ns'), n, ns) ∈ expand-ord φ⟧
    ⟹ f (n', ns') ≤ SPEC P
  shows f (n', ns') ≤ SPEC P
  using assms by (rule-tac assm-rule) auto

lemma expand-inv-upd-incoming:
  assumes expand-inv φ (n, ns)
    shows expand-inv-result φ (upd-incoming n ns)
using assms unfolding expand-inv-def upd-incoming-def by force


lemma upd-incoming-eq-old-next-pair:
  shows old-next-pair ‘ ns = old-next-pair ‘ (upd-incoming n ns) (is ?A = ?B)
proof
  { fix x
    let ?f = λn'. n'(|incoming := incoming n ∪ incoming n'|)
    assume xin:x∈old-next-pair ‘ ns
    then obtain n' where n'∈ns and xeq:x = (old n', next n') by auto
    {
      assume old n' = old n ∧ next n' = next n
      with ‹n'∈ns› 
      have ?f n'∈?f ‘ (ns ∩ {n'∈ns. old n' = old n ∧ next n' = next n})
        (is -∈?A)
        by auto
      hence old-next-pair (?f n')∈old-next-pair ‘ ?A
        by (rule-tac imageI) auto
      with xeq have x∈old-next-pair ‘ ?A by auto }
    moreover
    { assume old n' ≠ old n ∨ next n' ≠ next n (is ?cond n')
      with ‹n'∈ns› xeq
      have x∈old-next-pair ‘ (ns ∩ {n'∈ns. ?cond n'}) (is -∈?A)
        by auto
    }
    ultimately have
      x∈(old-next-pair ‘ (λn'. n'(|incoming := incoming n ∪ incoming n'|))
            ‘ (ns ∩ {n' ∈ ns. old n' = old n ∧ next n' = next n}))
          ∪ (old-next-pair
            ‘ (ns ∩ {n' ∈ ns. old n' ≠ old n ∨ next n' ≠ next n}))
      by fast
    hence x ∈ old-next-pair ‘ upd-incoming n ns
      using xin unfolding upd-incoming-def by auto
  }
  thus ?A ⊆ ?B by blast
```

next
show *?B* ⊆ *?A* **unfolding** *upd-incoming-def* **by** (*force intro*:*imageI*)
qed

lemma *expand-term-prop*:
expand-inv φ *n-ns*
⟹ *expand*$_T$ *n-ns* ≤ *SPEC* (λ(-, *nds*). *old-next-pair* ‘ *snd n-ns*⊆*old-next-pair* ‘*nds*
∧ *expand-inv-result* φ *nds*)
(**is** - ⟹ - ≤ *SPEC* (*?P n-ns*))
unfolding *expand*$_T$*-def*
apply (*rule-tac RECT-rule*[**where**
pre=λ(*n*, *ns*). *expand-inv* φ (*n*, *ns*) **and**
V=*expand-ord* φ
])
apply (*refine-mono*)
apply *simp*
apply *simp*
proof (*intro refine-vcg*)
case (*goal1* - - *n ns*)
have *old-next-pair* ‘ *ns* ⊆ *old-next-pair* ‘ (*upd-incoming n ns*)
by (*rule equalityD1*[*OF upd-incoming-eq-old-next-pair*])
thus *?case* **using** *expand-inv-upd-incoming*[*of* φ *n ns*] *goal1* **by** *auto*
next
case (*goal2 expand* - *n ns*)
let *?n′* = (|
name = *expand-new-name* (*name n*),
incoming = {*name n*},
new = *next n*,
old = {},
next = {}|)
let *?ns′* = {*n*} ∪ *ns*
have *SPEC-sub*:*SPEC* (*?P* (*?n′*, *?ns′*)) ≤ *SPEC* (*?P x*)
using *goal2* **by** (*rule-tac SPEC-rule*) *auto*
from *goal2* **have** *old-next-pair n*∉*old-next-pair* ‘ *ns* **by** *auto*
moreover hence *subset-next-pair*:
old-next-pair ‘ *ns* ⊂ *old-next-pair* ‘ (*insert n ns*)
by *auto*
ultimately have ((*?n′*, *?ns′*), (*n*, *ns*))∈*expand-ord* φ **using** *goal2*
by (*auto simp add*:*expand-ord-def expand-inv-def finite-psupset-def*)
moreover have *expand-inv* φ (*?n′*, *?ns′*)
using *goal2* **unfolding** *expand-inv-def* **by** *auto*
ultimately have *expand* (*?n′*, *?ns′*) ≤ *SPEC* (*?P* (*?n′*, *?ns′*))
using *goal2* **by** *fast*
with *SPEC-sub* **show** *?case* **apply** (*rule-tac order-trans*) **by** *fast*+
next
case *goal3* **thus** *?case* **by** (*auto simp add*:*expand-inv-def*)
next
case *goal4* **thus** *?case*

```
    apply (rule-tac expand-term-prop-help[OF expand-inv-impl])
    apply (simp add:expand-inv-def)+
    apply force
    done
next
  case goal5 thus ?case
    apply (rule-tac expand-term-prop-help[OF expand-inv-impl])
    apply (simp add:expand-inv-def)+
    apply force
    done
next
  case goal6 thus ?case by (simp add:expand-inv-def)
next
  case goal7 thus ?case
    apply (rule-tac expand-term-prop-help[OF expand-inv-impl])
    apply (simp add:expand-inv-def)+
    apply force
    done
next
  case goal8
  let ?n' = a(|
    new := new1 xa ∪ (new a − {xa}),
    old := insert xa (old a),
    next := next1 xa ∪ next a|)
  let ?n'' = λnm. a(|
    name := nm,
    new := new2 xa ∪ (new a − {xa}),
    old := insert xa (old a)|)
  have step:((?n', b), (a, b)) ∈ expand-ord φ ∧ expand-inv φ (?n', b)
    using goal8
    by (rule-tac expand-inv-impl) (auto simp add:expand-inv-def)
  hence assm1: f (?n', b) ≤ SPEC (?P (a, b))
    using goal8 by auto
  moreover
  { fix nm::node-name and nds::'a node set
    assume assm1: old-next-pair ‘ b ⊆ old-next-pair ‘ nds
      and assm2: expand-inv-result φ nds
    hence ((?n'' nm, nds), (a, b)) ∈ expand-ord φ
      ∧ expand-inv φ (?n'' nm, nds)
      using goal8 step
      by (rule-tac expand-inv-impl) auto
    hence f (?n'' nm, nds) ≤ SPEC (?P (?n'' nm, nds)) using goal8 by auto
    moreover
    have SPEC (?P (?n'' nm, nds)) ≤ SPEC (?P (a, b))
      using assm2 subset-trans[OF assm1] by auto
    ultimately have f (?n'' nm, nds) ≤ SPEC (?P (a, b))
      by (rule order-trans)
  }
  hence assm2: SPEC (?P (a, b))
```

```
    ≤ SPEC (λr. (case r of (nm, nds) ⇒ f (?n'' nm, nds)) ≤ SPEC (?P (a, b)))
    by (rule-tac SPEC-rule) auto
  from order-trans[OF assm1 assm2] show ?case using goal8 by auto
qed

lemma expand-eq-expand_T:
  assumes I: expand-inv φ n-ns
    shows expand_T n-ns = expand n-ns
  unfolding expand_T-def expand-def
  apply (rule RECT-eq-REC)
  apply refine-mono
  unfolding expand_T-def[symmetric]
  using expand-term-prop[OF I] by auto

lemma expand-nofail:
  assumes inv:expand-inv φ n-ns
    shows nofail (expand_T n-ns)
  using expand-term-prop[OF inv] by (simp add: pw-le-iff refine-pw-simps)

lemma [intro!]: expand-inv φ (
  (|
    name = expand-new-name expand-init,
    incoming = {expand-init},
    new = {φ},
    old = {},
    next = {} |),
  {})
by (auto simp add:expand-inv-def)

definition create-graph :: 'a frml ⇒ 'a node set nres
where
  create-graph φ ≡
    do {
      (-, nds) ← expand (
        (|
          name = expand-new-name expand-init,
          incoming = {expand-init},
          new = {φ},
          old = {},
          next = {}
        |)::'a node,
        {}::'a node set);
      RETURN nds
    }



definition create-graph_T :: 'a frml ⇒ 'a node set nres
where
```

```
  create-graph_T φ ≡ do {
    (-, nds) ← expand_T (
      ⦇
        name = expand-new-name expand-init,
        incoming = {expand-init},
        new = {φ},
        old = {},
        next = {}
      ⦈::'a node,
    {}::'a node set);
    RETURN nds
  }

lemma create-graph-eq-create-graph_T: create-graph φ = create-graph_T φ
unfolding create-graph_T-def create-graph-def
unfolding eq-iff
proof
  case goal1 thus ?case
    apply (refine-mono) unfolding expand-def expand_T-def
      by (rule REC-le-RECT, refine-mono)
next
  case goal2 thus ?case
     by (refine-mono, rule expand-eq-expand_T[unfolded eq-iff, THEN conjunct1])
auto
qed


lemma create-graph-finite: create-graph φ ≤ SPEC finite
proof −
  show ?thesis unfolding create-graph-def unfolding expand-def
    apply (intro refine-vcg)
    apply (rule-tac order-trans)
    apply (rule-tac REC-le-RECT, refine-mono)
    apply (fold expand_T-def)
    apply (rule-tac order-trans[OF expand-term-prop])
    apply auto[1]
    apply (rule-tac SPEC-rule)
    apply auto
  done
qed

lemma create-graph-nofail: nofail (create-graph φ)
by (rule-tac pwD1[OF create-graph-finite]) auto


abbreviation
  create-graph-rslt-exist ξ nds
  ≡ ∃ nd∈nds.
      expand-init∈incoming nd
```

$\wedge$ ($\forall \psi \in old\ nd.\ \xi \models_n \psi$) $\wedge$ ($\forall \psi \in next\ nd.\ \xi \models_n X_n\ \psi$)
$\wedge$ $\{\eta.\ \exists \mu.\ \mu\ U_n\ \eta \in old\ nd \wedge \xi \models_n \eta\} \subseteq old\ nd$

lemma *L4-7*:
assumes $\xi \models_n \varphi$
shows *create-graph* $\varphi \leq$ *SPEC* (*create-graph-rslt-exist* ξ)
using *assms* **unfolding** *create-graph-def*
by (*intro refine-vcg*, *rule-tac order-trans*, *rule-tac expand-prop-exist*) (
auto simp add:*expand-new-name-expand-init*)

lemma *expand-incoming-name-exist*:
assumes *name* (*fst n-ns*) > *expand-init*
$\wedge$ ($\forall$ *nm*$\in$*incoming* (*fst n-ns*). *nm* $\neq$ *expand-init* $\longrightarrow$ *nm*$\in$*name* ‘ (*snd n-ns*))
$\wedge$ *expand-assm-incoming n-ns* $\wedge$ *expand-name-ident* (*snd n-ns*) (**is** *?Q n-ns*)
and $\forall$ *nd*$\in$*snd n-ns*.
name nd > *expand-init*
$\wedge$ ($\forall$ *nm*$\in$*incoming nd*. *nm* $\neq$ *expand-init* $\longrightarrow$ *nm*$\in$*name* ‘ (*snd n-ns*))
(**is** *?P* (*snd n-ns*))
shows *expand n-ns* $\leq$ *SPEC* (λ*nm-nds*. *?P* (*snd nm-nds*))
using *assms*
proof (
rule-tac expand-rec-rule[**where** Φ=λ*n-ns*. *?Q n-ns* $\wedge$ *?P* (*snd n-ns*)],
simp,
intro refine-vcg)
case (*goal1 f x n ns*) **thus** *?case*
proof(*simp*, *clarify*)
case (*goal1* - - *nd*)
{ assume *nd*$\in$*ns*
hence *?case* **using** *goal1* **by** *auto* **}**
moreover
{ assume *nd*$\notin$*ns*
with *upd-incoming--elem*[*OF* ‹*nd*$\in$*upd-incoming n ns*›]
obtain *nd'* **where** *nd'*$\in$*ns* **and** *nd* = *nd'*(|*incoming* :=
incoming n $\cup$ *incoming nd'*|) $\wedge$
old nd' = *old n* $\wedge$
next nd' = *next n* **by** *auto*
hence *?case* **using** *goal1* **by** *auto* **}**
ultimately show *?case* **by** *fast*
qed
next
case (*goal2 f x n ns*)
hence *step*: $\bigwedge x.$ *?Q x* $\wedge$ *?P* (*snd x*) $\Longrightarrow$ *f x* $\leq$ *SPEC* ($\lambda x.$ *?P* (*snd x*))
and *QP*: *?Q* (*n*, *ns*) $\wedge$ *?P ns*
and *f-sup*: $\bigwedge x.$ *f x* $\leq$ *expand x* **by** *auto*
show *?case* **unfolding** ‹*x* = (*n*, *ns*)› **using** *QP expand-new-name-expand-init*
proof (*rule-tac step*)
case *goal1*
hence *name-less*: *name n* < *expand-new-name* (*name n*) **by** *auto*

```
      moreover hence ∀ nm∈incoming n. nm < expand-new-name (name n)
        using goal1 by auto
      moreover have ∀ q∈ns.
      name q < expand-new-name (name n) ∧
      (∀ nm∈incoming q.
          nm < expand-new-name (name n)) using name-less goal1 by force
      moreover have ∃!q'. (q' = n ∨ q' ∈ ns) ∧ name n = name q' using QP
        by force
      moreover have ∀ q∈ns.
     ∃!q'.
        (q' = n ∨ q' ∈ ns) ∧
        name q = name q'  using QP by auto
      ultimately show ?case using goal1  by simp
    qed
next
  case goal3 thus ?case by simp
next
  case goal4 thus ?case by simp
next
  case goal5 thus ?case by simp
next
  case goal6 thus ?case by simp
next
  case goal7 thus ?case by simp
next
  case (goal8 f x n ns ψ)
    hence goal-assms:
      ψ ∈ new n ∧ (∃ν μ. ψ = ν or_n μ ∨ ψ = ν U_n μ ∨ ψ = ν V_n μ)
      by (cases ψ) auto
    hence QP: ?Q (n, ns) ∧ ?P ns
      and step: ⋀x. ?Q x ∧ ?P (snd x) ⟹ f x ≤ SPEC (λx'. ?P (snd x'))
      and f-sup: ⋀x. f x ≤ expand x using goal8 by auto
    let ?x = (n(|new := new n − {ψ}, new := new1 ψ ∪ new (n(|new := new n
− {ψ}|)),
          old := {ψ} ∪ old (n(|new := new n − {ψ}|)),
          next := next1 ψ ∪ next (n(|new := new n − {ψ}|))|), ns)
    let ?props = λx r. expand-rslt-incoming r
      ∧ expand-rslt-name x r
      ∧ expand-name-ident (snd r)

    show ?case using goal-assms QP unfolding case-prod-unfold ‹x = (n, ns)›
    proof (rule-tac SPEC-rule-weak-nested2[where Q = ?props ?x])
      case goal1 thus ?case
        by (rule-tac order-trans, rule-tac f-sup, rule-tac expand-name-propag) simp
    next
      case goal2 thus ?case
       by (rule-tac SPEC-rule-param2[where P = λx r. ?P (snd r)], rule-tac step)
          auto
    next
```

```
    case (goal3 nm nds)
      thus ?case using goal3
      proof (rule-tac SPEC-rule-weak[
          where P = λx r. ?P (snd r)
          and Q = λx r. expand-rslt-exist-eq x r ∧ ?props x r])
        case goal1 thus ?case
          by (rule-tac SPEC-rule-conjI,
                rule-tac order-trans,
                rule-tac f-sup,
                rule-tac expand-rslt-exist-eq,
                rule-tac order-trans,
                rule-tac f-sup,
                rule-tac expand-name-propag) force
      next
        case goal2 thus ?case
          by (rule-tac SPEC-rule-param2[where P = λx r. ?P (snd r)],
            rule-tac step) force
      next
        case (goal3 nm' nds') thus ?case by simp
      qed
  qed
qed

lemma create-graph--incoming-name-exist:
  shows create-graph φ ≤ SPEC (λnds. ∀nd∈nds. expand-init < name nd ∧
(∀nm∈incoming nd. nm ≠ expand-init ⟶ nm∈name ‘ nds))
  using assms unfolding create-graph-def
  by (intro refine-vcg,
    rule-tac order-trans,
    rule-tac expand-incoming-name-exist) (
    auto simp add:expand-new-name-expand-init)


abbreviation
  expand-rslt-all--ex-equiv ξ nd nds ≡
   (∃nd'∈nds.
    name nd ∈ incoming nd'
    ∧ (∀ψ∈old nd'. suffix 1 ξ ⊨n ψ) ∧ (∀ψ∈next nd'. suffix 1 ξ ⊨n Xn ψ)
    ∧ {η. ∃μ. μ Un η ∈ old nd' ∧ suffix 1 ξ ⊨n η} ⊆ old nd')

abbreviation
  expand-rslt-all ξ n-ns nm-nds ≡
   (∀nd∈snd nm-nds. name nd ∉ name ‘ (snd n-ns) ∧
        (∀ψ∈old nd. ξ ⊨n ψ) ∧ (∀ψ∈next nd. ξ ⊨n Xn ψ)
        ⟶ expand-rslt-all--ex-equiv ξ nd (snd nm-nds))

lemma expand-prop-all:
  assumes expand-assm-incoming n-ns ∧ expand-name-ident (snd n-ns) (is ?Q
n-ns)
```

```
    shows expand n-ns ≤ SPEC (expand-rslt-all ξ n-ns)
      (is - ≤ SPEC (?P n-ns))
using assms
proof (rule-tac expand-rec-rule[where Φ=?Q], simp, intro refine-vcg)
  case goal1 thus ?case by (simp, rule-tac upd-incoming--ident) simp+
next
  case (goal2 f x n ns)
    hence step: ⋀x. ?Q x ⟹ f x ≤ SPEC (?P x)
      and Q: ?Q (n, ns)
      and f-sup: ⋀x. f x ≤ expand x by auto
    let ?x = (⦇name = expand-new-name (name n),
      incoming = {name n}, new = next n, old = {}, next = {}⦈, {n} ∪ ns)
    from Q have name-le: name n < expand-new-name (name n) by auto
    show ?case unfolding ‹x = (n, ns)›

    proof (rule-tac SPEC-rule-weak[where
        Q = λp r.
        (expand-assm-exist (suffix 1 ξ) ?x ⟶ expand-rslt-exist (suffix 1 ξ) ?x r)
        ∧ expand-rslt-exist-eq p r ∧ (expand-name-ident (snd r))])
      case goal1 thus ?case
      proof (rule-tac SPEC-rule-conjI,
            rule-tac order-trans,
            rule-tac f-sup,
            rule-tac expand-prop-exist,
            rule-tac SPEC-rule-conjI,
            rule-tac order-trans,
            rule-tac f-sup,
            rule-tac expand-rslt-exist-eq,
            rule-tac order-trans,
            rule-tac f-sup,
            rule-tac expand-name-propag--name-ident)
        case goal1 thus ?case using Q name-le by force
      qed
    next
      case goal2 thus ?case using Q name-le by (rule-tac step) force
    next
      case (goal3 nm nds)
        then obtain n' where n'∈nds
          and eq-node: expand-rslt-exist-eq--node n n' by auto
        with goal3 have ex1-name: ∃!q∈nds. name n = name q by auto
        hence nds-eq: nds = {n'} ∪ {x ∈ nds. name n ≠ name x}
          using eq-node ‹n'∈nds› by blast
        have name-notin: name n ∉ name ‘ ns using Q by auto
        have P-x: expand-rslt-all ξ ?x (nm, nds) using goal3 by fast

        show ?case unfolding snd-conv
        proof(clarify)
          fix nd
          assume nd ∈ nds and name-img: name nd ∉ name ‘ ns
```

```
          and nd-old-equiv: ∀ψ∈old nd. ξ ⊨n ψ
          and nd-next-equiv: ∀ψ∈next nd. ξ ⊨n X n ψ
        { assume name nd = name n
          with nds-eq eq-node ‹nd∈nds› have nd = n' by auto
          with goal3(1)[THEN conjunct1, simplified]
            nd-old-equiv nd-next-equiv eq-node
          have expand-rslt-all--ex-equiv ξ nd nds by simp }
        moreover
        { assume name nd ≠ name n
          with name-img ‹nd ∈ nds› nd-old-equiv nd-next-equiv P-x
          have expand-rslt-all--ex-equiv ξ nd nds by simp }
        ultimately show expand-rslt-all--ex-equiv ξ nd nds by fast
      qed
    qed
next
  case goal3 thus ?case by auto
next
  case (goal4 f)
    hence step: ⋀x. ?Q x ⟹ f x ≤ SPEC (?P x) by simp+
    show ?case using goal4 by (rule-tac SPEC-rule-param2, rule-tac step) auto
next
  case (goal5 f)
    hence step: ⋀x. ?Q x ⟹ f x ≤ SPEC (?P x) by simp+
    show ?case using goal5 by (rule-tac SPEC-rule-param2, rule-tac step) auto
next
  case goal6 thus ?case by auto
next
  case (goal7 f)
    hence step: ⋀x. ?Q x ⟹ f x ≤ SPEC (?P x) by simp+
    show ?case using goal7 by (rule-tac SPEC-rule-param2, rule-tac step) auto
next
  case (goal8 f x n ns ψ)
    hence goal-assms: ψ ∈ new n ∧ (∃ν μ. ψ = ν or_n μ ∨ ψ = ν U_n μ ∨ ψ = ν
V_n μ)
      by (cases ψ) auto
    have Q: ?Q (n, ns)
     and step: ⋀x. ?Q x ⟹ f x ≤ SPEC (?P x)
     and f-sup: ⋀x. f x ≤ expand x using goal8 by auto
    let ?x = (n(|new := new n − {ψ}, new := new1 ψ ∪ new (n(|new := new n
− {ψ}|)),
          old := {ψ} ∪ old (n(|new := new n − {ψ}|)),
                       next := next1 ψ ∪ next (n(|new := new n − {ψ}|)
        |), ns)
    let ?props = λx r. expand-rslt-incoming r
      ∧ expand-rslt-name x r
      ∧ expand-name-ident (snd r)

    show ?case using goal-assms Q unfolding case-prod-unfold ‹x = (n, ns)›
    proof (rule-tac SPEC-rule-weak-nested2[where Q = ?props ?x])
```

```
  case goal1 thus ?case by (rule-tac order-trans,
    rule-tac f-sup,
    rule-tac expand-name-propag) simp
next
  case goal2 thus ?case
    by (rule-tac SPEC-rule-param2[where P = ?P], rule-tac step) auto
next
  case (goal3 nm nds)
    thus ?case using goal3
    proof (rule-tac SPEC-rule-weak[where
        P = ?P and
        Q = λx r. expand-rslt-exist-eq x r ∧ ?props x r])
      case goal1 thus ?case
        by (rule-tac SPEC-rule-conjI,
              rule-tac order-trans,
              rule-tac f-sup,
              rule-tac expand-rslt-exist-eq,
              rule-tac order-trans,
              rule-tac f-sup,
              rule-tac expand-name-propag) auto
    next
      case goal2 thus ?case
        by (rule-tac SPEC-rule-param2[where P = ?P], rule-tac step) auto
    next
      case (goal3 nm' nds')
        hence P-x: ?P (n, ns) (nm, nds)
          and P-x': ?P (n, nds) (nm', nds') by simp+
        show ?case unfolding snd-conv
        proof(clarify)
          fix nd
          assume nd∈nds' and
            name-nd-notin: name nd ∉ name ‘ ns and
            old-equiv: ∀ψ∈old nd. ξ ⊨n ψ and
            next-equiv: ∀ψ∈next nd. ξ ⊨n Xn ψ

          { assume name nd ∈ name ‘ nds
            then obtain n' where n' ∈ nds and name nd = name n' by auto
            then obtain nd' where nd'∈nds'
              and nd'-eq: expand-rslt-exist-eq--node n' nd'
              using goal3 by auto
            moreover have ∀q∈nds'. ∃!q'∈nds'. name q = name q'
              using goal3 by simp
           ultimately have nd' = nd using ‹name nd = name n'› ‹nd ∈ nds'›
              by auto
            with nd'-eq have n'-eq: expand-rslt-exist-eq--node n' nd by simp
            hence name n'∉name ‘ ns
              and ∀ψ∈old n'. ξ ⊨n ψ and ∀ψ∈next n'. ξ ⊨n Xn ψ
              using name-nd-notin old-equiv next-equiv ‹n' ∈ nds›
              by auto
```

```
        hence expand-rslt-all--ex-equiv ξ n' nds
          (is ∃ nd'∈nds. ?sthm n' nd')
        using P-x ‹n' ∈ nds› unfolding snd-conv by blast
        then obtain sucnd where sucnd∈nds and sthm: ?sthm n' sucnd
          by blast
        moreover then obtain sucnd' where sucnd'∈nds' and
          sucnd'-eq: expand-rslt-exist-eq--node sucnd sucnd'
          using goal3 by auto
        ultimately have ?sthm n' sucnd' by auto
        hence expand-rslt-all--ex-equiv ξ nd nds' using ‹sucnd'∈nds'›
        unfolding ‹name nd = name n'› by blast }
      moreover
      { assume name nd ∉ name ' nds
        with ‹nd∈nds'› P-x' old-equiv next-equiv
        have expand-rslt-all--ex-equiv ξ nd nds'
          unfolding snd-conv by blast }
      ultimately show expand-rslt-all--ex-equiv ξ nd nds' by fast
    qed
  qed
 qed
qed

abbreviation
  create-graph-rslt-all ξ nds
  ≡ ∀ q∈nds. (∀ ψ∈old q. ξ ⊨n ψ) ∧ (∀ ψ∈next q. ξ ⊨n Xn ψ)
   ⟶ (∃ q'∈nds. name q ∈ incoming q'
       ∧ (∀ ψ∈old q'. suffix 1 ξ ⊨n ψ)
       ∧ (∀ ψ∈next q'. suffix 1 ξ ⊨n Xn ψ)
       ∧ {η. ∃ μ. μ Un η ∈ old q' ∧ suffix 1 ξ ⊨n η} ⊆ old q')

lemma L4-5:
  shows create-graph φ ≤ SPEC (create-graph-rslt-all ξ)
  unfolding create-graph-def
  by (intro refine-vcg,
    rule-tac order-trans,
    rule-tac expand-prop-all) (auto simp add:expand-new-name-expand-init)
```

5.4 Creation of GBA

This section formalizes the second part of the algorithm, that creates the actual generalized Büchi automata from the set of nodes.

```
definition create-gba-from-nodes
  :: 'a frml ⇒ 'a node set ⇒ ('a node, 'a set) gba-rec
where create-gba-from-nodes φ qs ≡ (|
  g-V = qs,
  g-E = {(q, q'). q∈qs ∧ q'∈qs ∧ name q∈incoming q'},
  g-V0 = {q∈qs. expand-init∈incoming q},
  gbg-F = {{q∈qs. μ Un η∈old q ⟶ η∈old q}|μ η. μ Un η ∈ subfrmlsn φ},
  gba-L = λq l. q∈qs ∧ {p. propn(p)∈old q}⊆l ∧ {p. npropn(p)∈old q} ∩ l = {}
```

⟧

end

locale *create-gba-from-nodes-precond* =
  fixes φ :: *'a ltln*
  fixes *qs* :: *'a node set*
  assumes *res*: *inres* (*create-graph* φ) *qs*
begin

  lemma *finite-qs*[*simp*, *intro*!]: *finite qs*
    using *res create-graph-finite* **by** (*auto simp add*: *refine-pw-simps pw-le-iff*)

  lemma *create-gba-from-nodes--invar*: *gba* (*create-gba-from-nodes* φ *qs*)
    using [[*simproc finite-Collect*]]
    apply *unfold-locales*
    apply (*auto*
      intro!: *finite-vimageI subfrmlsn-finite injI*
      simp: *create-gba-from-nodes-def*)
    done

 sublocale *gba*!: *gba create-gba-from-nodes* φ *qs* **using** *create-gba-from-nodes--invar*
.

  lemma *create-gba-from-nodes--fin*: *finite* (*g-V* (*create-gba-from-nodes* φ *qs*))
    unfolding *create-gba-from-nodes-def* **by** *auto*

  lemma *create-gba-from-nodes--ipath*:
    shows *ipath gba.E r*
      $\longleftrightarrow$ ($\forall$ *i*. *r i* $\in$*qs* $\wedge$ *name* (*r i*)$\in$*incoming* (*r* (*Suc i*)))
    unfolding *create-gba-from-nodes-def ipath-def*
    by *auto*

  lemma *create-gba-from-nodes--is-run*:
    shows *gba.is-run r*
          $\longleftrightarrow$ *expand-init* $\in$ *incoming* (*r 0*)
              $\wedge$ ($\forall$ *i*. *r i*$\in$*qs* $\wedge$ *name* (*r i*)$\in$*incoming* (*r* (*Suc i*)))
    unfolding *gba.is-run-def*
    apply (*simp add*: *create-gba-from-nodes--ipath*)
    apply (*auto simp*: *create-gba-from-nodes-def*)
    done

context begin interpretation *LTL-Syntax* **.**

abbreviation
  auto-run-j j ξ *q* $\equiv$
      ($\forall\,\psi$$\in$*old q*. *suffix j* ξ $\models_n$ ψ) $\wedge$ ($\forall\,\psi$$\in$*next q*. *suffix j* ξ $\models_n$ X_n ψ) $\wedge$
      {η. $\exists\,\mu$. μ U_n η $\in$ *old q* $\wedge$ *suffix j* ξ $\models_n$ η} $\subseteq$ *old q*

fun *auto-run* :: [*'a interprt*, *'a node set*] ⇒ *'a node word*
where
auto-run ξ nds 0
= (*SOME q*. *q*∈*nds* ∧ *expand-init* ∈ *incoming q* ∧ *auto-run-j 0 ξ q*)
| *auto-run ξ nds* (*Suc k*)
= (*SOME q'*. *q'*∈*nds* ∧ *name* (*auto-run ξ nds k*) ∈ *incoming q'*
∧ *auto-run-j* (*Suc k*) *ξ q'*)

lemma *run-propag-on-create-graph*:
assumes *ipath gba.E σ*
shows *σ k*∈*qs* ∧ *name* (*σ k*)∈*incoming* (*σ* (*k*+*1*))
using *assms*
by (*auto simp*: *create-gba-from-nodes--ipath*)

lemma *expand-false-propag*:
assumes *false*$_n$ ∉ *old* (*fst n-ns*) ∧ (∀ *nd*∈*snd n-ns*. *false*$_n$ ∉ *old nd*)
(**is** *?Q n-ns*)
shows *expand n-ns* ≤ *SPEC* (λ*nm-nds*. ∀ *nd*∈*snd nm-nds*. *false*$_n$ ∉ *old nd*)
using *assms*
proof (*rule-tac expand-rec-rule*[**where** Φ=*?Q*], *simp*, *intro refine-vcg*)
case *goal1* **thus** *?case* **by** (*simp*, *rule-tac upd-incoming--ident*) *auto*
next
case *goal8* **thus** *?case* **by** (*rule-tac SPEC-rule-nested2*) *auto*
qed *auto*

lemma *false-propag-on-create-graph*:
shows *create-graph φ* ≤ *SPEC* (λ*nds*. ∀ *nd*∈*nds*. *false*$_n$ ∉ *old nd*)
unfolding *create-graph-def*
by (*intro refine-vcg*, *rule-tac order-trans*, *rule-tac expand-false-propag*) *auto*

lemma *expand-and-propag*:
assumes *μ and*$_n$ *η* ∈ *old* (*fst n-ns*)
⟶ {*μ*, *η*} ⊆ *old* (*fst n-ns*) ∪ *new* (*fst n-ns*) (**is** *?Q n-ns*)
and ∀ *nd*∈*snd n-ns*. *μ and*$_n$ *η* ∈ *old nd* ⟶ {*μ*, *η*} ⊆ *old nd* (**is** *?P* (*snd n-ns*))
shows *expand n-ns* ≤ *SPEC* (λ*nm-nds*. *?P* (*snd nm-nds*))
using *assms*
proof (*rule-tac expand-rec-rule*[**where** Φ=λ*x*. *?Q x* ∧ *?P* (*snd x*)],
simp, *intro refine-vcg*)
case *goal1* **thus** *?case* **by** (*simp*, *rule-tac upd-incoming--ident*) *auto*
next
case (*goal4 f x n ns*)
hence *step*: ⋀*x*. *?Q x* ∧ *?P* (*snd x*) ⟹ *f x* ≤ *SPEC* (λ*x'*. *?P* (*snd x'*)) **by** *simp*
thus *?case* **using** *goal4* **by** (*rule-tac step*) *auto*
next

case (*goal5 f x n ns*)
hence *step*: $\bigwedge x.\ ?Q\ x \wedge ?P\ (snd\ x) \Longrightarrow f\ x \leq SPEC\ (\lambda x'.\ ?P\ (snd\ x'))$ **by** *simp*
thus *?case* **using** *goal5* **by** (*rule-tac step*) *auto*
next
case (*goal6 f x n ns*) **thus** *?case* **by** *auto*
next
case (*goal7 f x n ns*)
hence *step*: $\bigwedge x.\ ?Q\ x \wedge ?P\ (snd\ x) \Longrightarrow f\ x \leq SPEC\ (\lambda x'.\ ?P\ (snd\ x'))$ **by** *simp*
thus *?case* **using** *goal7* **by** (*rule-tac step*) *auto*
next
case (*goal8 f x n ns* ψ)
hence *goal-assms*: $\psi \in new\ n$
$\wedge \neg (\exists q.\ \psi = prop_n(q) \vee \psi = nprop_n(q))$
$\wedge\ \psi \neq true_n \wedge \psi \neq false_n$
$\wedge \neg (\exists \nu\ \mu.\ \psi = \nu\ and_n\ \mu \vee \psi = X_n\ \nu)$
$\wedge (\exists \nu\ \mu.\ \psi = \nu\ or_n\ \mu \vee \psi = \nu\ U_n\ \mu \vee \psi = \nu\ V_n\ \mu)$
by (*cases* ψ) *auto*
have *QP*: $?Q\ (n,\ ns) \wedge ?P\ ns$
and *step*: $\bigwedge x.\ ?Q\ x \wedge ?P\ (snd\ x) \Longrightarrow f\ x \leq SPEC\ (\lambda x'.\ ?P\ (snd\ x'))$
using *goal8* **by** *simp+*
show *?case* **using** *goal-assms QP* **unfolding** *case-prod-unfold* ‹$x = (n,\ ns)$›
proof (*rule-tac SPEC-rule-nested2*)
case *goal1* **thus** *?case* **by** (*rule-tac step*) *auto*
next
case *goal2* **thus** *?case* **by** (*rule-tac step*) *auto*
qed
qed *auto*

lemma *and-propag-on-create-graph*:
shows *create-graph* φ
$\leq SPEC\ (\lambda nds.\ \forall\, nd{\in}nds.\ \mu\ and_n\ \eta \in old\ nd \longrightarrow \{\mu,\ \eta\} \subseteq old\ nd)$
unfolding *create-graph-def*
by (*intro refine-vcg*, *rule-tac order-trans*, *rule-tac expand-and-propag*) *auto*

lemma *expand-or-propag*:
assumes $\mu\ or_n\ \eta \in old\ (fst\ n\text{-}ns)$
$\longrightarrow \{\mu,\ \eta\} \cap (old\ (fst\ n\text{-}ns) \cup new\ (fst\ n\text{-}ns)) \neq \{\}$ (**is** *?Q n-ns*)
and $\forall\, nd{\in}snd\ n\text{-}ns.\ \mu\ or_n\ \eta \in old\ nd \longrightarrow \{\mu,\ \eta\} \cap old\ nd \neq \{\}$
(**is** *?P* (*snd n-ns*))
shows *expand n-ns* $\leq SPEC\ (\lambda nm\text{-}nds.\ ?P\ (snd\ nm\text{-}nds))$
using *assms*
proof (*rule-tac expand-rec-rule*[**where** $\Phi{=}\lambda x.\ ?Q\ x \wedge ?P\ (snd\ x)$],
simp, *intro refine-vcg*)
case *goal1* **thus** *?case* **by** (*simp*, *rule-tac upd-incoming--ident*) *auto*
next
case (*goal4 f x n ns*)

hence *step*: $\bigwedge x.\ ?Q\ x \wedge ?P\ (snd\ x) \Longrightarrow f\ x \leq SPEC\ (\lambda x'.\ ?P\ (snd\ x'))$ **by** *simp*
thus *?case* **using** *goal4* **by** (*rule-tac step*) *auto*
next
case (*goal5 f x n ns*)
hence *step*: $\bigwedge x.\ ?Q\ x \wedge ?P\ (snd\ x) \Longrightarrow f\ x \leq SPEC\ (\lambda x'.\ ?P\ (snd\ x'))$ **by** *simp*
thus *?case* **using** *goal5* **by** (*rule-tac step*) *auto*
next
case (*goal6 f x n ns*) **thus** *?case* **by** *auto*
next
case (*goal7 f x n ns*)
hence *step*: $\bigwedge x.\ ?Q\ x \wedge ?P\ (snd\ x) \Longrightarrow f\ x \leq SPEC\ (\lambda x'.\ ?P\ (snd\ x'))$ **by** *simp*
thus *?case* **using** *goal7* **by** (*rule-tac step*) *auto*
next
case (*goal8 f x n ns* ψ)
hence *goal-assms*: $\psi \in new\ n$
$\wedge \neg\ (\exists\, q.\ \psi = prop_n(q) \vee \psi = nprop_n(q))$
$\wedge\ \psi \neq true_n \wedge \psi \neq false_n$
$\wedge \neg\ (\exists \nu\ \mu.\ \psi = \nu\ and_n\ \mu \vee \psi = X_n\ \nu)$
$\wedge\ (\exists \nu\ \mu.\ \psi = \nu\ or_n\ \mu \vee \psi = \nu\ U_n\ \mu \vee \psi = \nu\ V_n\ \mu)$
by (*cases* ψ) *auto*
have *QP*: $?Q\ (n,\ ns) \wedge ?P\ ns$
and *step*: $\bigwedge x.\ ?Q\ x \wedge ?P\ (snd\ x) \Longrightarrow f\ x \leq SPEC\ (\lambda x'.\ ?P\ (snd\ x'))$
using *goal8* **by** *simp+*
show *?case* **using** *goal-assms QP* **unfolding** *case-prod-unfold* ‹$x = (n,\ ns)$›
proof (*rule-tac SPEC-rule-nested2*)
case *goal1* **thus** *?case* **by** (*rule-tac step*) *auto*
next
case *goal2* **thus** *?case* **by** (*rule-tac step*) *auto*
qed
qed *auto*

lemma *or-propag-on-create-graph*:
shows *create-graph* φ
$\leq SPEC\ (\lambda nds.\ \forall\, nd{\in}nds.\ \mu\ or_n\ \eta \in old\ nd \longrightarrow \{\mu,\ \eta\} \cap old\ nd \neq \{\})$
unfolding *create-graph-def*
by (*intro refine-vcg*, *rule-tac order-trans*, *rule-tac expand-or-propag*) *auto*

abbreviation
next-propag--assm μ *n-ns* $\equiv$
$(X_n\ \mu \in old\ (fst\ n\text{-}ns) \longrightarrow \mu \in next\ (fst\ n\text{-}ns))$
$\wedge\ (\forall\, nd{\in}snd\ n\text{-}ns.\ X_n\ \mu \in old\ nd \wedge name\ nd \in incoming\ (fst\ n\text{-}ns)$
$\longrightarrow \mu \in old\ (fst\ n\text{-}ns) \cup new\ (fst\ n\text{-}ns))$

abbreviation
next-propag--rslt μ *ns* $\equiv$

$\forall\, nd{\in}ns.\ \forall\, nd'{\in}ns.\ X_n\ \mu \in old\ nd \wedge name\ nd \in incoming\ nd' \longrightarrow \mu \in old\ nd'$

lemma *expand-next-propag*:
fixes *n-ns* :: - × *'a node set*
assumes *next-propag--assm* μ *n-ns*
$\wedge$ *next-propag--rslt* μ (*snd n-ns*)
$\wedge$ *expand-assm-incoming n-ns*
$\wedge$ *expand-name-ident* (*snd n-ns*) (**is** *?Q n-ns*)
shows *expand n-ns* $\leq$ *SPEC* (λr. *next-propag--rslt* μ (*snd r*))
(**is** - $\leq$ *SPEC ?P*)
using *assms*
proof (*rule-tac expand-rec-rule*[**where** Φ=*?Q*], *simp*, *intro refine-vcg*)
case (*goal1 f x n ns*)
thus *?case*
proof (*simp*, *rule-tac upd-incoming--ident*)
case *goal1*
{ fix *nd* :: *'a node* **and** *nd'* :: *'a node*
let *?X-prop* = $X_n\ \mu \in old\ nd \wedge name\ nd \in incoming\ nd' \longrightarrow \mu \in old\ nd'$
assume *nd*$\in$*ns* **and** *nd'-elem*: *nd'*$\in$*upd-incoming n ns*
{ assume *nd'*$\in$*ns*
hence *?X-prop* **using** ‹*nd*$\in$*ns*› *goal1* **by** *auto* **}**
moreover
{ assume *nd'*$\notin$*ns* **and** *X-in-nd*: X_n μ*∈old nd* **and** *name nd* ∈*incoming nd'*
with *upd-incoming--elem*[*of nd' n ns*] *nd'-elem*
obtain *nd''* **where** *nd''*$\in$*ns*
and *nd'-eq*: *nd'* = *nd''*⦇*incoming* := *incoming n* ∪ *incoming nd''*⦈
and *old-eq*: *old nd''* = *old n* **by** *auto*
{ assume *name nd* ∈ *incoming n*
with *X-in-nd* ‹*nd*$\in$*ns*› **have** μ ∈ *old n* **using** *goal1* **by** *auto*
hence μ ∈ *old nd'* **using** *nd'-eq old-eq* **by** *simp* **}**
moreover
{ assume *name nd* ∉ *incoming n*
hence *name nd* ∈ *incoming nd''*
using ‹*name nd* ∈*incoming nd'*› *nd'-eq* **by** *simp*
hence μ ∈ *old nd'* **unfolding** *nd'-eq*
using ‹*nd*$\in$*ns*› ‹*nd''*$\in$*ns*› *X-in-nd goal1* **by** *auto* **}**
ultimately have μ ∈ *old nd'* **by** *fast* **}**
ultimately have *?X-prop* **by** *auto* **}**
thus *?case* **by** *auto*
next
case *goal2* **thus** *?case* **by** *simp*
qed
next
case (*goal2 f x n ns*)
hence *step*: $\bigwedge x.$ *?Q x* $\Longrightarrow$ *f x* $\leq$ *SPEC ?P*
and *f-sup*: $\bigwedge x.$ *f x* $\leq$ *expand x* **by** *auto*

```
    have Q: ?Q (n, ns) using goal2 by auto
    from Q have name-le: name n < expand-new-name (name n) by auto
    let ?x' = (⦇name = expand-new-name (name n),
                incoming = {name n}, new = next n,
                old = {}, next = {}⦈, {n} ∪ ns)
    have Q'1: expand-assm-incoming ?x' ∧ expand-name-ident (snd ?x')
    using ‹new n = {}› Q[THEN conjunct2, THEN conjunct2] name-le by force
    have Q'2: next-propag--assm μ ?x' ∧ next-propag--rslt μ (snd ?x')
      using Q ‹new n = {}› by auto

    show ?case using ‹new n = {}› unfolding ‹x = (n, ns)›

    proof (rule-tac SPEC-rule-weak[where
        Q = λ- r. expand-name-ident (snd r) and P = λ-. ?P])
      case goal1 thus ?case using Q'1
        by (rule-tac order-trans,
          rule-tac f-sup,
          rule-tac expand-name-propag--name-ident) auto
    next
      case goal2 thus ?case using Q'1 Q'2 by (rule-tac step) simp
    next
      case (goal3 nm nds) thus ?case by simp
    qed
next
  case goal3 thus ?case by auto
next
  case (goal4 f)
    hence step: ⋀x. ?Q x ⟹ f x ≤ SPEC ?P by simp+
    show ?case using goal4 by (rule-tac SPEC-rule-param2, rule-tac step) auto
next
  case (goal5 f)
    hence step: ⋀x. ?Q x ⟹ f x ≤ SPEC ?P by simp+
    show ?case using goal5 by (rule-tac SPEC-rule-param2, rule-tac step) auto
next
  case goal6 thus ?case by auto
next
  case (goal7 f)
    hence step: ⋀x. ?Q x ⟹ f x ≤ SPEC ?P by simp+
    show ?case using goal7 by (rule-tac SPEC-rule-param2, rule-tac step) auto
next
  case (goal8 f x n ns ψ)
    hence goal-assms: ψ ∈ new n ∧ (∃ν μ. ψ = ν or_n μ ∨ ψ = ν U_n μ ∨ ψ = ν
V_n μ)
      by (cases ψ) auto
    have Q: ?Q (n, ns)
     and step: ⋀x. ?Q x ⟹ f x ≤ SPEC ?P
     and f-sup: ⋀x. f x ≤ expand x using goal8 by auto
    let ?x = (n⦇new := new n − {ψ}, new := new1 ψ ∪ new (n⦇new := new n
− {ψ}⦈),
```

```
      old := {ψ} ∪ old (n(|new := new n − {ψ}|)),
                        next := next1 ψ ∪ next (n(|new := new n − {ψ}|))
      |), ns)
  let ?props = λx r. expand-rslt-exist-eq x r
    ∧ expand-rslt-incoming r ∧ expand-rslt-name x r ∧ expand-name-ident (snd
r)

  show ?case using goal-assms Q unfolding case-prod-unfold ‹x = (n, ns)›

  proof (rule-tac SPEC-rule-weak-nested2[where Q = ?props ?x])
    case goal1 thus ?case
      by (rule-tac SPEC-rule-conjI,
          rule-tac order-trans,
          rule-tac f-sup,
          rule-tac expand-rslt-exist-eq,
          rule-tac order-trans,
          rule-tac f-sup,
          rule-tac expand-name-propag) simp+
  next
    case goal2 thus ?case
      by (rule-tac SPEC-rule-param2[where P = λ-. ?P], rule-tac step) auto
  next
    case (goal3 nm nds)
      let ?x' = (n(|new := new n − {ψ},
        name := fst (nm, nds),
        new := new2 ψ ∪ new (n(|new := new n − {ψ}|)),
        old := {ψ} ∪ old (n(|new := new n − {ψ}|))|), nds)
      show ?case using goal3
      proof (rule-tac step)
        case goal1
          hence expand-assm-incoming ?x' by auto
          moreover
          have nds-ident: expand-name-ident (snd ?x') using goal1 by simp
          moreover
          have X_n μ ∈ old (fst ?x') ⟶ μ∈next (fst ?x')
          using Q[THEN conjunct1] goal-assms by auto
          moreover
          have next-propag--rslt μ (snd ?x') using goal1 by simp
          moreover
          have name-nds-eq:
            name ‘ nds = name ‘ ns ∪ name ‘ {nd∈nds. name nd ≥ name n}
            using goal1 by auto
          have ∀ nd∈nds. (X_n μ ∈ old nd ∧ name nd ∈ incoming (fst ?x'))
                        ⟶ μ∈old (fst ?x')∪new (fst ?x')
           (is ∀ nd∈nds. ?assm (fst ?x') nd ⟶ ?concl (fst ?x') nd)
          proof
            fix nd
            assume nd∈nds
            { assume loc-assm: name nd∈name ‘ ns
```

then obtain n' **where** $n'\in ns$ **and** *name* $n' =$ *name nd* **by** *auto*
moreover then obtain nd' **where** $nd'\in nds$
and n'-nd'-*eq*: *expand-rslt-exist-eq--node* n' nd'
using *goal1* **by** *auto*
ultimately have $nd = nd'$
using *nds-ident* ‹$nd\in nds$› *loc-assm* **by** *auto*
moreover from *goal1* **have** *?assm n* $n' \longrightarrow$ *?concl n* n'
using ‹$n'\in ns$› **by** *auto*
ultimately have *?assm* (*fst ?x'*) *nd* $\longrightarrow$ *?concl* (*fst ?x'*) *nd*
using n'-nd'-*eq* **by** *auto* **}**
moreover
{ assume *name nd*$\notin$*name* ‘ *ns*
with *name-nds-eq* ‹$nd\in nds$› **have** *name nd* $\geq$ *name n* **by** *auto*
hence ¬ (*?assm* (*fst ?x'*) *nd*) **using** *goal1* **by** *auto* **}**
ultimately show *?assm* (*fst ?x'*) *nd* $\longrightarrow$ *?concl* (*fst ?x'*) *nd* **by** *auto*
qed
ultimately show *?case* **by** *simp*
qed
qed
qed

lemma *next-propag-on-create-graph*:
shows *create-graph* φ
$\leq$ *SPEC* ($\lambda nds.\ \forall\, n\in nds.\ \forall\, n'\in nds.\ X_n\ \mu\in old\ n \wedge name\ n\in incoming\ n' \longrightarrow \mu\in old\ n'$)
unfolding *create-graph-def*
by (*intro refine-vcg*,
rule-tac order-trans,
rule-tac expand-next-propag) (*auto simp add*:*expand-new-name-expand-init*)

abbreviation
release-propag--assm $\mu\ \eta$ *n-ns* $\equiv$
($\mu\ V_n\ \eta \in old$ (*fst n-ns*)
$\longrightarrow \{\mu, \eta\}\subseteq old$ (*fst n-ns*)$\cup new$ (*fst n-ns*) $\vee$
($\eta\in old$ (*fst n-ns*)$\cup new$ (*fst n-ns*)) $\wedge\ \mu\ V_n\ \eta \in next$ (*fst n-ns*))
$\wedge$ ($\forall\, nd\in snd$ *n-ns*.
$\mu\ V_n\ \eta \in old\ nd \wedge name\ nd \in incoming$ (*fst n-ns*)
$\longrightarrow \{\mu, \eta\}\subseteq old\ nd \vee$
($\eta\in old\ nd \wedge\ \mu\ V_n\ \eta \in old$ (*fst n-ns*)$\cup new$ (*fst n-ns*)))

abbreviation
release-propag--rslt $\mu\ \eta$ *ns* $\equiv$
$\forall\, nd\in ns$.
$\forall\, nd'\in ns$.
$\mu\ V_n\ \eta \in old\ nd \wedge name\ nd \in incoming\ nd'$
$\longrightarrow \{\mu, \eta\}\subseteq old\ nd \vee$
($\eta\in old\ nd \wedge\ \mu\ V_n\ \eta \in old\ nd'$)

```
lemma expand-release-propag:
  fixes n-ns :: - × 'a node set
  assumes release-propag--assm μ η n-ns
          ∧ release-propag--rslt μ η (snd n-ns)
          ∧ expand-assm-incoming n-ns
          ∧ expand-name-ident (snd n-ns) (is ?Q n-ns)
    shows expand n-ns ≤ SPEC (λr. release-propag--rslt μ η (snd r))
      (is - ≤ SPEC ?P)
using assms
proof (rule-tac expand-rec-rule[where Φ=?Q], simp, intro refine-vcg)
  case (goal1 f x n ns)
    thus ?case
    proof (simp, rule-tac upd-incoming--ident)
      case goal1
        { fix nd :: 'a node and nd' :: 'a node
          let ?V-prop = μ V_n η ∈ old nd ∧ name nd ∈ incoming nd'
            ⟶ {μ, η} ⊆ old nd ∨ η ∈ old nd ∧ μ V_n η ∈ old nd'
          assume nd∈ns and nd'-elem: nd'∈upd-incoming n ns
          { assume nd'∈ns
            hence ?V-prop using ‹nd∈ns› goal1 by auto }
          moreover
          { assume nd'∉ns
            and V-in-nd: μ V_n η ∈ old nd and name nd ∈incoming nd'
            with upd-incoming--elem[of nd' n ns] nd'-elem
            obtain nd'' where nd''∈ns
              and nd'-eq: nd' = nd''(|incoming := incoming n ∪ incoming nd''|)
              and old-eq: old nd'' = old n
              by auto
            { assume name nd ∈ incoming n
              with V-in-nd ‹nd∈ns›
              have {μ, η} ⊆ old nd ∨ η ∈ old nd ∧ μ V_n η ∈ old n
                using goal1 by auto
              hence {μ, η} ⊆ old nd ∨ η ∈ old nd ∧ μ V_n η ∈ old nd'
                using nd'-eq old-eq by simp }
            moreover
            { assume name nd ∉ incoming n
              hence name nd ∈ incoming nd''
                using ‹name nd ∈incoming nd'› nd'-eq by simp
              hence {μ, η} ⊆ old nd ∨ η ∈ old nd ∧ μ V_n η ∈ old nd'
                unfolding nd'-eq
                using ‹nd∈ns› ‹nd''∈ns› V-in-nd goal1 by auto }
            ultimately have {μ, η} ⊆ old nd ∨ η ∈ old nd ∧ μ V_n η ∈ old nd'
              by fast }
          ultimately have ?V-prop by auto }
        thus ?case by auto
    next
      case goal2 thus ?case by simp
    qed
next
```

```
  case (goal2 f x n ns)
    hence step: ⋀x. ?Q x ⟹ f x ≤ SPEC ?P
      and f-sup: ⋀x. f x ≤ expand x by auto
    have Q: ?Q (n, ns) using goal2 by auto
    from Q have name-le: name n < expand-new-name (name n) by auto
    let ?x' = (⦇name = expand-new-name (name n),
                incoming = {name n}, new = next n,
                old = {}, next = {}⦈, {n} ∪ ns)
    have Q'1: expand-assm-incoming ?x' ∧ expand-name-ident (snd ?x')
    using ‹new n = {}› Q[THEN conjunct2, THEN conjunct2] name-le by force
    have Q'2: release-propag--assm μ η ?x' ∧ release-propag--rslt μ η (snd ?x')
      using Q ‹new n = {}› by auto

    show ?case using ‹new n = {}› unfolding ‹x = (n, ns)›

    proof (rule-tac SPEC-rule-weak[where
        Q = λ- r. expand-name-ident (snd r) and P = λ-. ?P])
      case goal1 thus ?case using Q'1
        by (rule-tac order-trans,
          rule-tac f-sup,
          rule-tac expand-name-propag--name-ident) auto
    next
      case goal2 thus ?case using Q'1 Q'2 by (rule-tac step) simp
    next
      case (goal3 nm nds) thus ?case by simp
    qed
next
  case goal3 thus ?case by auto
next
  case (goal4 f x n ns ψ)
    hence goal-assms: ψ ∈ new n ∧ (∃ q. ψ = prop_n(q) ∨ ψ = nprop_n(q)) by simp
    have step: ⋀x. ?Q x ⟹ f x ≤ SPEC ?P
      and Q: ?Q (n, ns) using goal4 by simp+
   show ?case using Q goal-assms by (rule-tac SPEC-rule-param2, rule-tac step)
auto
next
  case (goal5 f x n ns ψ)
    hence goal-assms: ψ ∈ new n ∧ ψ = true_n by simp
    have step: ⋀x. ?Q x ⟹ f x ≤ SPEC ?P
      and Q: ?Q (n, ns) using goal5 by simp+
   show ?case using Q goal-assms by (rule-tac SPEC-rule-param2, rule-tac step)
auto
next
  case goal6 thus ?case by auto
next
  case (goal7 f x n ns ψ)
    hence goal-assms: ψ ∈ new n ∧ (∃ν μ. ψ = ν and_n μ ∨ ψ = X_n ν) by simp
    have step: ⋀x. ?Q x ⟹ f x ≤ SPEC ?P
      and Q: ?Q (n, ns) using goal7 by simp+
```

```
  show ?case using Q goal-assms by (rule-tac SPEC-rule-param2, rule-tac step)
auto
next
 case (goal8 f x n ns ψ)
  hence goal-assms: ψ ∈ new n ∧ (∃ν μ. ψ = ν or_n μ ∨ ψ = ν U_n μ ∨ ψ = ν
V_n μ)
   by (cases ψ) auto
  have Q: ?Q (n, ns)
   and step: ⋀x. ?Q x ⟹ f x ≤ SPEC ?P
   and f-sup: ⋀x. f x ≤ expand x using goal8 by auto
  let ?x = (n(|new := new n − {ψ}, new := new1 ψ ∪ new (n(|new := new n
− {ψ}|)),
       old := {ψ} ∪ old (n(|new := new n − {ψ}|)),
                    next := next1 ψ ∪ next (n(|new := new n − {ψ}|)
    |), ns)
  let ?props = λx r. expand-rslt-exist-eq x r
    ∧ expand-rslt-incoming r ∧ expand-rslt-name x r ∧ expand-name-ident (snd
r)

  show ?case using goal-assms Q unfolding case-prod-unfold ‹x = (n, ns)›

  proof (rule-tac SPEC-rule-weak-nested2[where Q = ?props ?x])
   case goal1 thus ?case
    by (rule-tac SPEC-rule-conjI,
          rule-tac order-trans,
          rule-tac f-sup,
          rule-tac expand-rslt-exist-eq,
          rule-tac order-trans,
          rule-tac f-sup,
          rule-tac expand-name-propag) simp+
  next
   case goal2 thus ?case
    by (rule-tac SPEC-rule-param2[where P = λ-. ?P], rule-tac step) auto
  next
   case (goal3 nm nds)
    let ?x' = (n(|new := new n − {ψ},
      name := fst (nm, nds),
      new := new2 ψ ∪ new (n(|new := new n − {ψ}|)),
      old := {ψ} ∪ old (n(|new := new n − {ψ}|))|), nds)
    show ?case using goal3
    proof (rule-tac step)
     case goal1
      hence expand-assm-incoming ?x' by auto
      moreover
      have nds-ident: expand-name-ident (snd ?x') using goal1 by simp
      moreover
      have (μ V_n η ∈ old (fst ?x')
          ⟶ ({μ, η}⊆old (fst ?x') ∪ new (fst ?x')
            ∨ (η∈old (fst ?x')∪new (fst ?x')
```

```
                    ∧ μ V_n η ∈ next (fst ?x')))
          using Q[THEN conjunct1] goal-assms by auto
          moreover
          have release-propag--rslt μ η (snd ?x') using goal1 by simp
          moreover
          have name-nds-eq:
            name ‘ nds = name ‘ ns ∪ name ‘ {nd∈nds. name nd ≥ name n}
            using goal1 by auto
          have ∀ nd∈nds. (μ V_n η ∈ old nd ∧ name nd ∈ incoming (fst ?x'))
                    ⟶ ({μ, η}⊆old nd
                        ∨ (η∈old nd ∧ μ V_n η ∈old (fst ?x')∪new (fst ?x')))
           (is ∀ nd∈nds. ?assm (fst ?x') nd ⟶ ?concl (fst ?x') nd)
          proof
            fix nd
            assume nd∈nds
            { assume loc-assm: name nd∈name ‘ ns
              then obtain n' where n'∈ns and name n' = name nd by auto
              moreover then obtain nd' where nd'∈nds
                and n'-nd'-eq: expand-rslt-exist-eq--node n' nd'
                using goal1 by auto
              ultimately have nd = nd' using nds-ident ‹nd∈nds› loc-assm
                by auto
              moreover from goal1 have ?assm n n' ⟶ ?concl n n'
                using ‹n'∈ns› by auto
              ultimately have ?assm (fst ?x') nd ⟶ ?concl (fst ?x') nd
                using n'-nd'-eq by auto }
            moreover
            { assume name nd∉name ‘ ns
              with name-nds-eq ‹nd∈nds› have name nd ≥ name n by auto
              hence ¬ (?assm (fst ?x') nd) using goal1 by auto }
            ultimately show ?assm (fst ?x') nd ⟶ ?concl (fst ?x') nd by auto
          qed
          ultimately show ?case by simp
        qed
      qed
    qed

lemma release-propag-on-create-graph:
  shows create-graph φ
        ≤ SPEC (λnds. ∀ n∈nds. ∀ n'∈nds. μ V_n η∈old n ∧ name n∈incoming n'
                                  ⟶ ({μ, η}⊆old n ∨ η∈old n ∧ μ V_n η∈old n'))
  unfolding create-graph-def
  by (intro refine-vcg,
    rule-tac order-trans,
    rule-tac expand-release-propag) (auto simp add:expand-new-name-expand-init)


abbreviation
  until-propag--assm f g n-ns ≡
```

$(f\ U_n\ g \in old\ (fst\ n\text{-}ns)$
$\longrightarrow (g \in old\ (fst\ n\text{-}ns) \cup new\ (fst\ n\text{-}ns)$
$\vee\ (f \in old\ (fst\ n\text{-}ns) \cup new\ (fst\ n\text{-}ns) \wedge f\ U_n\ g \in next\ (fst\ n\text{-}ns))))$
$\wedge\ (\forall\, nd \in snd\ n\text{-}ns.\ f\ U_n\ g \in old\ nd \wedge name\ nd \in incoming\ (fst\ n\text{-}ns)$
$\longrightarrow (g \in old\ nd \vee (f \in old\ nd \wedge f\ U_n\ g \in old\ (fst\ n\text{-}ns) \cup new\ (fst\ n\text{-}ns))))$

abbreviation
until-propag--rslt f g ns $\equiv$
$\forall\, n \in ns.\ \forall\, nd \in ns.\ f\ U_n\ g \in old\ n \wedge name\ n \in incoming\ nd$
$\longrightarrow (g \in old\ n \vee (f \in old\ n \wedge f\ U_n\ g \in old\ nd))$

lemma *expand-until-propag*:
fixes *n-ns* :: - $\times$ *'a node set*
assumes *until-propag--assm* $\mu\ \eta$ *n-ns*
$\wedge$ *until-propag--rslt* $\mu\ \eta$ (*snd n-ns*)
$\wedge$ *expand-assm-incoming n-ns*
$\wedge$ *expand-name-ident* (*snd n-ns*) (**is** *?Q n-ns*)
shows *expand n-ns* $\leq$ *SPEC* (λr. *until-propag--rslt* $\mu\ \eta$ (*snd r*))
(**is** - $\leq$ *SPEC ?P*)
using *assms*
proof (*rule-tac expand-rec-rule*[**where** Φ=*?Q*], *simp*, *intro refine-vcg*)
case (*goal1 f x n ns*)
thus *?case*
proof (*simp*, *rule-tac upd-incoming--ident*)
case *goal1*
{ fix *nd* :: *'a node* **and** *nd'* :: *'a node*
let *?U-prop* = $\mu\ U_n\ \eta \in old\ nd \wedge name\ nd \in incoming\ nd'$
$\longrightarrow \eta \in old\ nd \vee \mu \in old\ nd \wedge \mu\ U_n\ \eta \in old\ nd'$
assume *nd*$\in$*ns* **and** *nd'-elem*: *nd'*$\in$*upd-incoming n ns*
{ assume *nd'*$\in$*ns*
hence *?U-prop* **using** ‹*nd*$\in$*ns*› *goal1* **by** *auto* **}**
moreover
{ assume *nd'*$\notin$*ns* **and**
U-in-nd: $\mu\ U_n\ \eta \in old\ nd$ **and** *name nd* $\in$*incoming nd'*
with *upd-incoming--elem*[*of nd' n ns*] *nd'-elem*
obtain *nd''* **where** *nd''*$\in$*ns*
and *nd'-eq*: *nd'* = *nd''*(|*incoming* := *incoming n* $\cup$ *incoming nd''*|)
and *old-eq*: *old nd''* = *old n* **by** *auto*
{ assume *name nd* $\in$ *incoming n*
with *U-in-nd* ‹*nd*$\in$*ns*›
have $\eta \in old\ nd \vee \mu \in old\ nd \wedge \mu\ U_n\ \eta \in old\ n$ **using** *goal1* **by** *auto*
hence $\eta \in old\ nd \vee \mu \in old\ nd \wedge \mu\ U_n\ \eta \in old\ nd'$
using *nd'-eq old-eq* **by** *simp* **}**
moreover
{ assume *name nd* $\notin$ *incoming n*
hence *name nd* $\in$ *incoming nd''*
using ‹*name nd* $\in$*incoming nd'*› *nd'-eq* **by** *simp*
hence $\eta \in old\ nd \vee \mu \in old\ nd \wedge \mu\ U_n\ \eta \in old\ nd'$ **unfolding** *nd'-eq*
using ‹*nd*$\in$*ns*› ‹*nd''*$\in$*ns*› *U-in-nd goal1* **by** *auto* **}**

```
        ultimately have η ∈ old nd ∨ μ ∈ old nd ∧ μ U_n η ∈ old nd' by fast }
       ultimately have ?U-prop by auto }
     thus ?case by auto
   next
    case goal2 thus ?case by simp
   qed
next
 case (goal2 f x n ns)
   hence step: ⋀x. ?Q x ⟹ f x ≤ SPEC ?P
    and f-sup: ⋀x. f x ≤ expand x by auto
   have Q: ?Q (n, ns) using goal2 by auto
   from Q have name-le: name n < expand-new-name (name n) by auto
   let ?x' = (⦇name = expand-new-name (name n),
              incoming = {name n}, new = next n,
              old = {}, next = {}⦈, {n} ∪ ns)
   have Q'1: expand-assm-incoming ?x' ∧ expand-name-ident (snd ?x')
   using ‹new n = {}› Q[THEN conjunct2, THEN conjunct2] name-le by force
   have Q'2: until-propag--assm μ η ?x' ∧ until-propag--rslt μ η (snd ?x')
    using Q ‹new n = {}› by auto

   show ?case using ‹new n = {}› unfolding ‹x = (n, ns)›

   proof (rule-tac SPEC-rule-weak[where
      Q = λ- r. expand-name-ident (snd r) and P = λ-. ?P])
    case goal1 thus ?case using Q'1
     by (rule-tac order-trans,
       rule-tac f-sup,
       rule-tac expand-name-propag--name-ident) auto
   next
    case goal2 thus ?case using Q'1 Q'2 by (rule-tac step) simp
   next
    case (goal3 nm nds) thus ?case by simp
   qed
next
 case goal3 thus ?case by auto
next
 case (goal4 f)
   hence step: ⋀x. ?Q x ⟹ f x ≤ SPEC ?P by simp+
   show ?case using goal4 by (rule-tac SPEC-rule-param2, rule-tac step) auto
next
 case (goal5 f)
   hence step: ⋀x. ?Q x ⟹ f x ≤ SPEC ?P by simp+
   show ?case using goal5 by (rule-tac SPEC-rule-param2, rule-tac step) auto
next
 case goal6 thus ?case by auto
next
 case (goal7 f)
   hence step: ⋀x. ?Q x ⟹ f x ≤ SPEC ?P by simp+
   show ?case using goal7 by (rule-tac SPEC-rule-param2, rule-tac step) auto
```

next
case (*goal8 f x n ns* ψ)
hence *goal-assms*: $\psi \in new\ n \wedge (\exists \nu\ \mu.\ \psi = \nu\ or_n\ \mu \vee \psi = \nu\ U_n\ \mu \vee \psi = \nu\ V_n\ \mu)$
by (*cases* ψ) *auto*
have *Q*: *?Q* (*n*, *ns*)
and *step*: $\bigwedge x.$ *?Q x* $\Longrightarrow$ *f x* $\leq$ *SPEC ?P*
and *f-sup*: $\bigwedge x.$ *f x* $\leq$ *expand x* **using** *goal8* **by** *auto*
let *?x* = (*n*(|*new* := *new n* − {ψ}, *new* := *new1* ψ ∪ *new* (*n*(|*new* := *new n* − {ψ}|)),
old := {ψ} ∪ *old* (*n*(|*new* := *new n* − {ψ}|)),
next := *next1* ψ ∪ *next* (*n*(|*new* := *new n* − {ψ}|)
|), *ns*)
let *?props* = λ*x r*. *expand-rslt-exist-eq x r*
∧ *expand-rslt-incoming r* ∧ *expand-rslt-name x r* ∧ *expand-name-ident* (*snd r*)

show *?case* **using** *goal-assms Q* **unfolding** *case-prod-unfold* ‹*x* = (*n*, *ns*)›

proof (*rule-tac SPEC-rule-weak-nested2*[**where** *Q* = *?props ?x*])
case *goal1* **thus** *?case*
by (*rule-tac SPEC-rule-conjI*,
rule-tac order-trans,
rule-tac f-sup,
rule-tac expand-rslt-exist-eq,
rule-tac order-trans,
rule-tac f-sup,
rule-tac expand-name-propag) *simp+*
next
case *goal2* **thus** *?case*
by (*rule-tac SPEC-rule-param2*[**where** *P* = λ-. *?P*], *rule-tac step*) *auto*
next
case (*goal3 nm nds*)
let *?x′* = (*n*(|*new* := *new n* − {ψ},
name := *fst* (*nm*, *nds*),
new := *new2* ψ ∪ *new* (*n*(|*new* := *new n* − {ψ}|)),
old := {ψ} ∪ *old* (*n*(|*new* := *new n* − {ψ}|))|), *nds*)
show *?case* **using** *goal3*
proof (*rule-tac step*)
case *goal1*
hence *expand-assm-incoming ?x′* **by** *auto*
moreover
have *nds-ident*: *expand-name-ident* (*snd ?x′*) **using** *goal1* **by** *simp*
moreover
have ($\mu\ U_n\ \eta \in old$ (*fst ?x′*)
$\longrightarrow$ ($\eta \in old$ (*fst ?x′*) ∪ *new* (*fst ?x′*)
∨ ($\mu \in old$ (*fst ?x′*) ∪ *new* (*fst ?x′*)
∧ $\mu\ U_n\ \eta \in next$ (*fst ?x′*))))
using *Q*[*THEN conjunct1*] *goal-assms* **by** *auto*

```
          moreover
          have until-propag--rslt μ η (snd ?x') using goal1 by simp
          moreover
          have name-nds-eq:
            name ‘ nds = name ‘ ns ∪ name ‘ {nd∈nds. name nd ≥ name n}
            using goal1 by auto
          have ∀ nd∈nds. (μ U_n η ∈ old nd ∧ name nd ∈ incoming (fst ?x'))
            ⟶ (η∈old nd ∨ (μ∈old nd ∧ μ U_n η ∈old (fst ?x')∪new (fst ?x')))
           (is ∀ nd∈nds. ?assm (fst ?x') nd ⟶ ?concl (fst ?x') nd)
          proof
            fix nd
            assume nd∈nds
            { assume loc-assm: name nd∈name ‘ ns
              then obtain n' where n'∈ns and name n' = name nd by auto
              moreover then obtain nd' where nd'∈nds
                and n'-nd'-eq: expand-rslt-exist-eq--node n' nd'
                using goal1 by auto
              ultimately have nd = nd'
                using nds-ident ‹nd∈nds› loc-assm by auto
              moreover from goal1 have ?assm n n' ⟶ ?concl n n'
                using ‹n'∈ns› by auto
              ultimately have ?assm (fst ?x') nd ⟶ ?concl (fst ?x') nd
                using n'-nd'-eq by auto }
            moreover
            { assume name nd∉name ‘ ns
              with name-nds-eq ‹nd∈nds› have name nd ≥ name n by auto
              hence ¬ (?assm (fst ?x') nd) using goal1 by auto }
            ultimately show ?assm (fst ?x') nd ⟶ ?concl (fst ?x') nd by auto
          qed
          ultimately show ?case by simp
        qed
      qed
qed

lemma until-propag-on-create-graph:
  shows create-graph φ
        ≤ SPEC (λnds. ∀ n∈nds. ∀ n'∈nds. μ U_n η∈old n ∧ name n∈incoming n'
  ⟶ (η∈old n ∨ μ∈old n ∧ μ U_n η∈old n'))
  unfolding create-graph-def
  by (intro refine-vcg,
    rule-tac order-trans,
    rule-tac expand-until-propag) (auto simp add:expand-new-name-expand-init)


definition all-subfrmls :: 'a node ⇒ 'a frml set
where
  all-subfrmls n ≡ ⋃(subfrmlsn ‘ (new n ∪ old n ∪ next n))

lemma all-subfrmls--UnionD:
```

assumes $(\bigcup x{\in}A.\ subfrmlsn\ x) \subseteq B$
and $x{\in}A$ **and** $y{\in}subfrmlsn\ x$
shows $y{\in}B$
proof −
note *subfrmlsn-id*[*of* x]
also have $subfrmlsn\ x \subseteq (\bigcup x{\in}A.\ subfrmlsn\ x)$ **using** *assms* **by** *auto*
finally show *?thesis* **using** *assms* **by** *auto*
qed

lemma *expand-all-subfrmls-propag*:
assumes $all\text{-}subfrmls\ (fst\ n\text{-}ns) \subseteq B$
$\wedge\ (\forall\, nd{\in}snd\ n\text{-}ns.\ all\text{-}subfrmls\ nd \subseteq B)$ (**is** *?Q n-ns*)
shows $expand\ n\text{-}ns \leq SPEC\ (\lambda r.\ \forall\, nd{\in}snd\ r.\ all\text{-}subfrmls\ nd \subseteq B)$
(**is** - $\leq SPEC\ ?P$)
using *assms*
proof (*rule-tac expand-rec-rule*[**where** Φ=*?Q*], *simp*, *intro refine-vcg*)
case *goal1* **thus** *?case*
proof (*simp*, *rule-tac upd-incoming--ident*)
case *goal1* **thus** *?case* **by** *auto*
next
case *goal2* **thus** *?case* **by** (*simp add*:*all-subfrmls-def*)
qed
next
case *goal2* **thus** *?case* **by** (*auto simp add*:*all-subfrmls-def*)
next
case *goal3* **thus** *?case* **by** (*auto simp add*:*all-subfrmls-def*)
next
case (*goal4 f*)
hence *step*: $\bigwedge x.\ ?Q\ x \Longrightarrow f\ x \leq SPEC\ ?P$ **by** *simp+*
show *?case* **using** *goal4* **by** (*rule-tac step*) (*auto simp add*:*all-subfrmls-def*)
next
case (*goal5 f - n ns* ψ)
hence *goal-assms*: $\psi \in new\ n \wedge \psi = true_n$ **by** *simp*
have *step*: $\bigwedge x.\ ?Q\ x \Longrightarrow f\ x \leq SPEC\ ?P$
and *Q*: *?Q* (*n*, *ns*) **using** *goal5* **by** *simp+*
show *?case* **using** *Q goal-assms*
by (*rule-tac step*) (*auto dest*:*all-subfrmls--UnionD simp add*:*all-subfrmls-def*)
next
case *goal6* **thus** *?case* **by** *auto*
next
case (*goal7 f x n ns* ψ)
hence *goal-assms*: $\psi \in new\ n \wedge (\exists \nu\ \mu.\ \psi = \nu\ and_n\ \mu \vee \psi = X_n\ \nu)$ **by** *simp*
have *step*: $\bigwedge x.\ ?Q\ x \Longrightarrow f\ x \leq SPEC\ ?P$
and *Q*: *?Q* (*n*, *ns*) **using** *goal7* **by** *simp+*
show *?case* **using** *Q goal-assms*
by (*rule-tac step*) (*auto dest*:*all-subfrmls--UnionD simp add*:*all-subfrmls-def*)
next
case (*goal8 f x n ns* ψ)

```
    hence goal-assms: ψ ∈ new n
      ∧ ¬ (∃ q. ψ = prop_n(q) ∨ ψ = nprop_n(q))
      ∧ ψ ≠ true_n ∧ ψ ≠ false_n
      ∧  ¬ (∃ν μ. ψ = ν and_n μ ∨ ψ = X_n ν)
      ∧ (∃ν μ. ψ = ν or_n μ ∨ ψ = ν U_n μ ∨ ψ = ν V_n μ)
      by (cases ψ) auto
    have Q: ?Q (n, ns) and step: ⋀x. ?Q x ⟹ f x ≤ SPEC ?P
      using goal8 by simp+
    show ?case using goal-assms Q unfolding case-prod-unfold ‹x = (n, ns)›
    proof (rule-tac SPEC-rule-nested2)
      case goal1 thus ?case
        by (rule-tac step) (auto
          dest:all-subfrmls--UnionD
          simp add:all-subfrmls-def)
    next
      case goal2 thus ?case
        by (rule-tac step) (auto
          dest:all-subfrmls--UnionD
          simp add:all-subfrmls-def)
    qed
qed

lemma old-propag-on-create-graph:
  shows create-graph φ ≤ SPEC (λnds. ∀ n∈nds. old n ⊆ subfrmlsn φ)
  unfolding create-graph-def
  by (intro refine-vcg,
    rule-tac order-trans,
    rule-tac expand-all-subfrmls-propag[where B = subfrmlsn φ])
   (force simp add:all-subfrmls-def expand-new-name-expand-init)+

lemma L4-2--aux:
  assumes run: ipath gba.E σ
      and μ U_n η ∈ old (σ 0)
      and ∀ j. (∀ i<j. {μ, μ U_n η} ⊆ old (σ i)) ⟶ η ∉ old (σ j)
    shows ∀ i. {μ, μ U_n η} ⊆ old (σ i) ∧ η ∉ old (σ i)
proof −
  { fix j
    have ∀ i<j. {μ, μ U_n η} ⊆ old (σ i) (is ?sbthm j)
    proof (induct j)
      show ?sbthm 0 by auto
    next
      fix k
      assume step: ?sbthm k
      hence σ-k-prop: η ∉ old (σ k)
        ∧ σ k∈qs ∧ σ (Suc k)∈qs
        ∧ name (σ k) ∈ incoming (σ (Suc k))
      using assms run-propag-on-create-graph[OF run] by auto
     with inres-SPEC[OF res until-propag-on-create-graph[where μ = μ and η =
η]]
```

```
    have {μ, μ U_n η} ⊆ old (σ k) (is ?subsetthm)
    proof (cases k)
      assume k = 0
      with assms σ-k-prop
        inres-SPEC[OF res until-propag-on-create-graph[where μ = μ and η =
η]]
      show ?subsetthm by auto
    next
      fix l
      assume k = Suc l
      hence {μ, μ U_n η}⊆old (σ l) ∧ η∉old (σ l)
        ∧ σ l∈qs ∧ σ k∈qs
        ∧ name (σ l)∈incoming (σ k)
      using step assms run-propag-on-create-graph[OF run] by auto
      with inres-SPEC[OF res until-propag-on-create-graph[where μ = μ and η
= η]]
      have μ U_n η∈old (σ k) by auto
      with σ-k-prop
        inres-SPEC[OF res until-propag-on-create-graph[where μ = μ and η =
η]]
      show ?subsetthm by auto
    qed
    with step show ?sbthm (Suc k) by (metis less-SucE)
  qed }
 with assms show ?thesis by auto
qed

lemma L4-2a:
 assumes ipath gba.E σ
    and μ U_n η ∈ old (σ 0)
  shows (∀ i. {μ, μ U_n η} ⊆ old (σ i) ∧ η ∉ old (σ i))
        ∨ (∃ j. (∀ i<j. {μ, μ U_n η} ⊆ old (σ i)) ∧ η ∈ old (σ j))
    (is ?A ∨ ?B)
proof −
 { assume ¬ ?B
  hence ?A using assms by (rule-tac L4-2--aux) blast+
  hence ?A ∨ ?B by fast }
 moreover
 { assume ?B
  hence ?A ∨ ?B by blast }
 ultimately show ?thesis by fast
qed

lemma L4-2b:
 assumes run: ipath gba.E σ
    and μ U_n η ∈ old (σ 0)
    and ACC: gba.is-acc σ
  shows ∃ j. (∀ i<j. {μ, μ U_n η} ⊆ old (σ i)) ∧ η ∈ old (σ j)
proof(rule ccontr)
```

```
  assume ¬ ?thesis
  hence contr: ∀ i. {μ, μ U_n η}⊆old(σ i) ∧ η∉old(σ i)
    using assms L4-2a[of σ μ η] by blast


  def S≡{q ∈ qs. μ U_n η ∈ old q ⟶ η ∈ old q}

  from assms inres-SPEC[OF res old-propag-on-create-graph]
    create-gba-from-nodes--ipath
  have μ U_n η ∈ subfrmlsn φ by (metis assms(2) set-mp)
  hence S∈gbg-F(create-gba-from-nodes φ qs)
    unfolding S-def create-gba-from-nodes-def by auto
  with ACC have 1: ∃_∞i. σ i ∈ S unfolding gba.is-acc-def by blast

  from INFM-EX[OF 1]
  obtain k where σ k ∈ qs and μ U_n η ∈ old (σ k) ⟶ η ∈ old (σ k)
    unfolding S-def by auto

  moreover have {μ, μ U_n η}⊆old(σ k) ∧ η∉old(σ k) using contr by auto
  ultimately show False by auto
qed

lemma L4-2c:
  assumes run: ipath gba.E σ
      and μ V_n η ∈ old (σ 0)
    shows ∀ i. η ∈ old (σ i) ∨ (∃ j<i. μ ∈ old (σ j))
proof −
  { fix i
    have {η, μ V_n η} ⊆ old (σ i) ∨ (∃ j<i. μ ∈ old (σ j)) (is ?thm i)
    proof (induct i)
      case 0
        have σ 0∈qs ∧ σ 1∈qs ∧ name (σ 0) ∈ incoming (σ 1)
        using create-gba-from-nodes--ipath assms by auto
        thus ?case
          using assms inres-SPEC[OF res release-propag-on-create-graph, of μ η]
          by auto
    next
      case (Suc k)
        note ‹?thm k›
        moreover
        { assume {η, μ V_n η} ⊆ old (σ k)
          moreover
          have σ k∈qs ∧ σ (Suc k)∈qs ∧ name (σ k) ∈ incoming (σ (Suc k))
          using create-gba-from-nodes--ipath assms by auto
          ultimately have μ ∈ old (σ k) ∨ μ V_n η ∈ old (σ (Suc k))
            using assms inres-SPEC[OF res release-propag-on-create-graph, of μ η]
            by auto
          moreover
          { assume μ ∈ old (σ k)
```

```
        hence ?case by blast }
      moreover
      { assume μ V_n η ∈ old (σ (Suc k))
        moreover
        have σ (Suc k)∈qs ∧ σ (Suc (Suc k))∈qs
          ∧ name (σ (Suc k)) ∈ incoming (σ (Suc (Suc k)))
        using create-gba-from-nodes--ipath assms by auto
        ultimately have ?case
          using assms
            inres-SPEC[OF res release-propag-on-create-graph, of μ η]
          by auto }
      ultimately have ?case by fast }
    moreover
    { assume ∃j<k. μ ∈ old (σ j)
      hence ?case by auto }
    ultimately show ?case by auto
  qed }
 thus ?thesis by auto
qed

lemma L4-8':
 assumes ipath gba.E σ (is ?inf-run σ)
 and gba.is-acc σ (is ?gbarel-accept σ)
 and ∀ i. gba.L (σ i) (ξ i) (is ?lgbarel-accept ξ σ)
 and ψ ∈ old (σ 0)
 shows ξ ⊨_n ψ
using assms proof (induct ψ arbitrary: σ ξ)
 case LTLnTrue show ?case by auto
next
 case LTLnFalse thus ?case
  using inres-SPEC[OF res false-propag-on-create-graph]
    create-gba-from-nodes--ipath
  by (metis)
next
 case (LTLnProp p) thus ?case
  unfolding create-gba-from-nodes-def by auto
next
 case (LTLnNProp p) thus ?case unfolding create-gba-from-nodes-def by auto
next
 case (LTLnAnd μ η) thus ?case
  using inres-SPEC[OF res and-propag-on-create-graph, of μ η]
    create-gba-from-nodes--ipath
    by (metis insert-subset ltln-semantics.simps(5))
next
 case (LTLnOr μ η)
  hence μ ∈ old (σ 0) ∨ η ∈ old (σ 0)
    using inres-SPEC[OF res or-propag-on-create-graph, of μ η]
    create-gba-from-nodes--ipath
    by (metis (full-types) Int-empty-left Int-insert-left-if0)
```

```
    moreover
    { assume μ ∈ old (σ 0)
      with LTLnOr have ξ ⊨n μ by auto }
    moreover
    { assume η ∈ old (σ 0)
      with LTLnOr have ξ ⊨n η by auto }
    ultimately show ?case by auto
next
  case (LTLnNext μ)
    with create-gba-from-nodes--ipath[of σ]
    have σ 0 ∈ qs ∧ σ 1 ∈ qs ∧ name (σ 0) ∈ incoming (σ 1) by auto
    with inres-SPEC[OF res next-propag-on-create-graph, of μ]
    have μ∈old (suffix 1 σ 0) using LTLnNext by auto
    moreover
    have ?inf-run (suffix 1 σ) and ?gbarel-accept (suffix 1 σ)
     and ?lgbarel-accept (suffix 1 ξ) (suffix 1 σ )
    using LTLnNext create-gba-from-nodes--ipath
    apply −
    apply (metis ipath-suffix)
    apply (auto simp del: suffix-nth) []
    apply (auto) []
    done
    ultimately show ?case using LTLnNext by simp
next
  case (LTLnUntil μ η)
    hence ∃j. (∀ i<j. {μ, μ U_n η} ⊆ old (σ i)) ∧ η ∈ old (σ j)
    using L4-2b by auto
    then obtain j where
      σ-pre: ∀ i<j. {μ, μ U_n η} ⊆ old (σ i) and η ∈ old (suffix j σ 0)
      by auto
    moreover have ?inf-run (suffix j σ) and ?gbarel-accept (suffix j σ)
            and ?lgbarel-accept (suffix j ξ) (suffix j σ )
      unfolding limit-suffix
      using LTLnUntil create-gba-from-nodes--ipath
      apply −
      apply (metis ipath-suffix)
      apply (auto simp del: suffix-nth) []
      apply (auto) []
      done
    ultimately have suffix j ξ ⊨n η using LTLnUntil by simp
    moreover
    { fix i
      assume i<j
      have ?inf-run (suffix i σ) and ?gbarel-accept (suffix i σ)
       and ?lgbarel-accept (suffix i ξ) (suffix i σ ) unfolding limit-suffix
      using LTLnUntil create-gba-from-nodes--ipath
      apply −
      apply (metis ipath-suffix)
      apply (auto simp del: suffix-nth) []
```

```
      apply (auto) []
      done
      moreover
      have μ∈old (suffix i σ 0) using σ-pre ‹i<j› by auto
      ultimately have suffix i ξ ⊨n μ using LTLnUntil by simp }
    ultimately show ?case by auto
next
  case (LTLnRelease μ η)
    { fix i
      assume η ∈ old (σ i) ∨ (∃j<i. μ ∈ old (σ j))
      moreover
      { assume η ∈ old (σ i)
        moreover
        have ?inf-run (suffix i σ) and ?gbarel-accept (suffix i σ)
         and ?lgbarel-accept (suffix i ξ) (suffix i σ ) unfolding limit-suffix
         using LTLnRelease create-gba-from-nodes--ipath
        apply −
        apply (metis ipath-suffix)
        apply (auto simp del: suffix-nth) []
        apply (auto) []
        done
        ultimately have suffix i ξ ⊨n η using LTLnRelease by auto }
      moreover
      { assume ∃j<i. μ ∈ old (σ j)
        then obtain j where j<i and μ ∈ old (σ j) by auto
        moreover
        have ?inf-run (suffix j σ) and ?gbarel-accept (suffix j σ)
         and ?lgbarel-accept (suffix j ξ) (suffix j σ ) unfolding limit-suffix
         using LTLnRelease create-gba-from-nodes--ipath
        apply −
        apply (metis ipath-suffix)
        apply (auto simp del: suffix-nth) []
        apply (auto) []
        done
        ultimately have suffix j ξ ⊨n μ using LTLnRelease by auto
        hence ∃j<i. suffix j ξ ⊨n μ using ‹j<i› by auto }
      ultimately have suffix i ξ ⊨n η ∨ (∃j<i. suffix j ξ ⊨n μ) by auto }
    thus ?case using LTLnRelease L4-2c by auto
qed


lemma L4-8:
  assumes gba.is-acc-run σ
      and ∀ i. gba.L (σ i) (ξ i)
      and ψ ∈ old (σ 0)
    shows ξ ⊨n ψ
  using assms
  unfolding gba.is-acc-run-def gba.is-run-def
  using L4-8′ by blast
```

```
lemma expand-expand-init-propag:
  assumes (∀ nm∈incoming n'. nm < name n')
          ∧ name n' ≤ name (fst n-ns)
          ∧ (incoming n' ∩ incoming (fst n-ns) ≠ {}
             ⟶ new n' ⊆ old (fst n-ns) ∪ new (fst n-ns))
           (is ?Q n-ns)
  and ∀ nd∈snd n-ns. ∀ nm∈incoming n'. nm∈incoming nd ⟶ new n' ⊆ old nd
    (is ?P (snd n-ns))
  shows expand n-ns ≤ SPEC (λr. name n'≤fst r ∧ ?P (snd r))
  using assms
proof (rule-tac expand-rec-rule[where Φ=λx. ?Q x ∧ ?P (snd x)],
    simp,
    intro refine-vcg)
  case (goal1 f x n ns)
    hence goal-assms: new n = {} ∧ ?Q (n, ns) ∧ ?P ns by simp
    { fix nd nm
      assume nd∈upd-incoming n ns and nm∈incoming n' and nm∈incoming nd
      { assume nd∈ns
        with goal-assms ‹nm∈incoming n'› ‹nm∈incoming nd› have new n' ⊆ old
nd
          by auto }
      moreover
      { assume nd∉ns
        with upd-incoming--elem[OF ‹nd∈upd-incoming n ns›]
        obtain nd' where nd'∈ns
                    and nd-eq: nd = nd'(|incoming := incoming n ∪ incoming nd'|)
                    and old-next-eq: old nd' = old n ∧ next nd' = next n by auto
        { assume nm∈incoming nd'
          with goal-assms ‹nm∈incoming n'› ‹nd'∈ns› have new n' ⊆ old nd
            unfolding nd-eq by auto }
        moreover
        { assume nm∈incoming n
          with nd-eq old-next-eq goal-assms ‹nm∈incoming n'›
          have new n' ⊆ old nd
            by auto }
        ultimately have new n' ⊆ old nd using ‹nm∈incoming nd› nd-eq by auto
}
      ultimately have new n' ⊆ old nd by fast }
    thus ?case using goal1 by auto
next
  case (goal2 f x n ns)
    hence goal-assms: new n = {} ∧ ?Q (n, ns) ∧ ?P ns
      and step: ⋀x. ?Q x ∧ ?P (snd x)
        ⟹ f x ≤ SPEC (λr. name n' ≤ fst r ∧ ?P (snd r))
      by simp+

    thus ?case
    proof (rule-tac step)
```

```
      case goal1
       have expand-new-name-less: name n < expand-new-name (name n) by auto
        moreover have name n ∉ incoming n' using goal-assms by auto
        ultimately show ?case using goal1 by auto
    qed
next
  case goal3 thus ?case by auto
next
  case (goal4 f x n ns)
    hence step: ⋀x. ?Q x ∧ ?P (snd x)
      ⟹ f x ≤ SPEC (λr. name n' ≤ fst r ∧ ?P (snd r))
      by simp+
    show ?case using goal4 by (rule-tac step) auto
next
  case (goal5 f x n ns)
    hence step: ⋀x. ?Q x ∧ ?P (snd x)
      ⟹ f x ≤ SPEC (λr. name n' ≤ fst r ∧ ?P (snd r))
      by simp+
    show ?case using goal5 by (rule-tac step) auto
next
  case goal6 thus ?case by auto
next
  case (goal7 f x n ns)
    hence step:
     ⋀x. ?Q x ∧ ?P (snd x) ⟹ f x ≤ SPEC (λr. name n' ≤ fst r ∧ ?P (snd r))
      by simp+
    show ?case using goal7 by (rule-tac step) auto
next
  case (goal8 f x n ns ψ)
    hence goal-assms: ψ ∈ new n
      ∧ ¬ (∃ q. ψ = prop_n(q) ∨ ψ = nprop_n(q))
      ∧ ψ ≠ true_n ∧ ψ ≠ false_n
      ∧ ¬ (∃ν μ. ψ = ν and_n μ ∨ ψ = X_n ν)
      ∧ (∃ν μ. ψ = ν or_n μ ∨ ψ = ν U_n μ ∨ ψ = ν V_n μ)
      by (cases ψ) auto
    have QP: ?Q (n, ns) ∧ ?P ns and
      step: ⋀x. ?Q x ∧ ?P (snd x)
      ⟹ f x ≤ SPEC (λr. name n' ≤ fst r ∧ ?P (snd r))
      using goal8 by simp+
    show ?case using goal-assms QP unfolding case-prod-unfold ‹x = (n, ns)›
    proof (rule-tac SPEC-rule-nested2)
      case goal1 thus ?case by (rule-tac step) auto
    next
      case (goal2 nm nds) thus ?case by (rule-tac step) auto
    qed
qed

lemma expand-init-propag-on-create-graph:
  shows create-graph φ
```

```
  ≤ SPEC(λnds. ∀ nd∈nds. expand-init∈incoming nd ⟶ φ ∈ old nd)
unfolding create-graph-def
by (intro refine-vcg, rule-tac order-trans,
    rule-tac expand-expand-init-propag[where
  n' = (| name = expand-new-name expand-init,
          incoming = {expand-init},
          new = {φ},
          old = {},
          next = {} |)]) (auto simp add:expand-new-name-expand-init)

lemma L4-6:
  assumes q∈gba.V0
    shows φ∈old q
using assms inres-SPEC[OF res expand-init-propag-on-create-graph]
unfolding create-gba-from-nodes-def by auto

lemma L4-9:
  assumes acc: gba.accept ξ
  shows ξ ⊨n φ
proof −
  from acc obtain σ where accept: gba.is-acc-run σ
    ∧ (∀ i. gba.L (σ i) (ξ i))
  unfolding gba.accept-def by auto
  hence σ 0∈gba.V0
    unfolding gba.is-acc-run-def gba.is-run-def by simp
  with L4-6 have φ∈old (σ 0) by auto
  with L4-8 accept show ?thesis by auto
qed

lemma L4-10:
  assumes ξ ⊨n φ
    shows gba.accept ξ
proof −
  def σ ≡ auto-run ξ qs
  let ?G = create-graph φ

  have σ-prop-0: (σ 0)∈qs ∧ expand-init∈incoming(σ 0) ∧ auto-run-j 0 ξ (σ 0)
    (is ?sbthm)
  using inres-SPEC[OF res L4-7[OF ‹ξ ⊨n φ›]]
  unfolding σ-def auto-run.simps by (rule-tac someI-ex, simp) blast

  have σ-valid: ∀ j. σ j ∈ qs ∧ auto-run-j j ξ (σ j) (is ∀ j. ?σ-valid j)
  proof
    fix j
    show ?σ-valid j
    proof(induct j)
      from σ-prop-0 show ?σ-valid 0 by fast
    next
      fix k
```

```
    assume goal-assms: σ k ∈ qs ∧ auto-run-j k ξ (σ k)
    with inres-SPEC[OF res L4-5, of suffix k ξ]
     have sbthm: ∃ q'. q'∈qs ∧ name (σ k)∈incoming q' ∧ auto-run-j (Suc k) ξ
q'
      (is ∃ q'. ?sbthm q')
      by auto
    have ?sbthm (σ (Suc k)) using someI-ex[where, OF sbthm]
    unfolding σ-def auto-run.simps by blast
    thus ?σ-valid (Suc k) by fast
  qed
 qed

 have σ-prop-Suc: ⋀k. σ k∈ qs ∧ σ (Suc k)∈qs
  ∧ name (σ k)∈incoming (σ (Suc k))
  ∧ auto-run-j (Suc k) ξ (σ (Suc k))
 proof −
  fix k
  from σ-valid have σ k ∈ qs and auto-run-j k ξ (σ k) by blast+
  with inres-SPEC[OF res L4-5, of suffix k ξ]
  have sbthm: ∃ q'. q'∈qs ∧ name (σ k)∈incoming q'
   ∧ auto-run-j (Suc k) ξ q' (is ∃ q'. ?sbthm q')
   by auto
  show σ k∈ qs ∧ ?sbthm (σ (Suc k)) using ‹σ k∈qs› someI-ex[where, OF
sbthm]
   unfolding σ-def auto-run.simps by blast
 qed

 have σ-vnaccpt:
     ∀ k μ η. μ U_n η ∈ old (σ k)
       ⟶ ¬ (∀ i. {μ, μ U_n η} ⊆ old (σ (k+i)) ∧
             η ∉ old (σ (k+i)))
 proof(clarify)
  fix k μ η
  assume U-in: μ U_n η ∈ old (σ k)
   and cntr-prm: ∀ i. {μ, μ U_n η} ⊆ old (σ (k+i)) ∧
                 η ∉ old (σ (k+i))

  have suffix k ξ ⊨_n μ U_n η using U-in σ-valid by force
  then obtain i
       where suffix (k+i) ξ ⊨_n η
        and ∀ j<i. suffix (k+j) ξ ⊨_n μ by auto
  moreover
  have μ U_n η ∈ old (σ (k+i)) ∧
       η ∉ old (σ (k+i)) using cntr-prm by auto
  ultimately show False using σ-valid by force
 qed

 have σ-exec: gba.is-run σ using σ-prop-0 σ-prop-Suc σ-valid
  unfolding gba.is-run-def ipath-def
```

```
  unfolding create-gba-from-nodes-def
  by auto
moreover
have σ-vaccpt:
    ∀ k μ η. μ U_n η ∈ old (σ k) ⟶
          (∃ j. (∀ i<j. {μ, μ U_n η} ⊆ old (σ (k+i))) ∧
                η ∈ old (σ (k+j)))
proof(clarify)
  fix k μ η
  assume U-in: μ U_n η ∈ old (σ k)
  hence ¬ (∀ i. {μ, μ U_n η} ⊆ old (suffix k σ i) ∧ η ∉ old (suffix k σ i))
  using σ-vnaccpt[THEN allE, of k] by auto
  moreover have suffix k σ 0 ∈ qs using σ-valid by auto
  ultimately
  show ∃ j. (∀ i<j. {μ, μ U_n η} ⊆ old (σ (k+i))) ∧ η ∈ old (σ (k+j))
    apply −
    apply (rule make-pos-rule'[OF L4-2a])
    apply (fold suffix-def)
    apply (rule ipath-suffix)
    using σ-exec[unfolded gba.is-run-def]
    apply simp

    using U-in apply simp

    apply simp
    done
qed

have gba.is-acc σ
  unfolding gba.is-acc-def
proof
  fix S
  assume S∈gba.F
  then obtain μ η where
       S-eq: S = {q ∈ qs. μ U_n η ∈ old q ⟶ η ∈ old q}
    and μ U_n η ∈ subfrmlsn φ
    by (auto simp add: create-gba-from-nodes-def)

  have range-subset: range σ ⊆ qs
  proof
    fix q
    assume q∈range σ
    with full-SetCompr-eq[of σ] obtain k where q = σ k by auto
    thus q ∈ qs using σ-valid by auto
  qed
  with limit-nonempty[of σ]
       limit-in-range[of σ]
       finite-subset[OF range-subset]
       inres-SPEC[OF res create-graph-finite]
```

```
  obtain q where q-in-limit: q ∈ limit σ
         and q-is-node: q∈qs by auto


  thus ∃∞i. σ i ∈ S
  proof(cases μ U_n η ∈ old q)
    assume μ U_n η ∉ old q
    with S-eq q-in-limit q-is-node
    show ?thesis
      by (auto simp: limit-iff-frequent intro: INFM-mono)
  next
    assume μ U_n η ∈ old q
    obtain k where q-eq: q = σ k using q-in-limit
    unfolding limit-iff-frequent by (metis (lifting, full-types) INFM-nat-le)

    have ∃∞ k. η ∈ old (σ k) unfolding INFM-nat
    proof(rule ccontr)
      assume ¬ (∀ m. ∃ n>m. η ∈ old (σ n))
      then obtain m
           where ∀ n>m. η ∉ old (σ n) by auto
      moreover
      from q-eq q-in-limit limit-iff-frequent[of q σ]
          INFM-nat[of λn. σ n = q]
      obtain n where m<n
                and σn-eq: σ n = σ k by auto
      moreover
      obtain j
       where η ∈ old (σ (n+j))
       using  σ-vaccpt ‹μ U_n η ∈ old q› unfolding q-eq
         by (fold σn-eq) force

      ultimately show False by auto
    qed

    hence ∃∞ k. σ k ∈ qs ∧ η ∈ old (σ k)
      using σ-valid by (auto intro:INF-mono)
    thus ∃∞ k. σ k ∈ S unfolding S-eq by (rule INFM-mono) simp
  qed
qed
moreover
have ∀ i. gba.L (σ i) (ξ i) (is ∀ i. ?sbthm i)
proof
  fix i
  from σ-valid have [simp]: σ i ∈ qs by auto
  have ∀ ψ∈old (σ i). suffix i ξ ⊨_n ψ using σ-valid by auto
  thus ?sbthm i unfolding create-gba-from-nodes-def by auto
qed

ultimately show ?thesis
```

unfolding *gba.accept-def gba.is-acc-run-def* **by** *blast*
qed

end
end

lemma *create-graph--name-ident*:
shows *create-graph* $\varphi \leq$ *SPEC* (λ*nds*. $\forall\, q{\in}$*nds*. $\exists!\, q'{\in}$*nds*. *name* q = *name* q')
using *assms* **unfolding** *create-graph-def*
by (*intro refine-vcg*,
rule-tac order-trans,
rule-tac expand-name-propag--name-ident)
(*auto simp add*:*expand-new-name-expand-init*)

corollary *create-graph--name-inj*:
shows *create-graph* $\varphi \leq$ *SPEC* (λ*nds*. *inj-on name nds*)
apply (*rule order-trans*[*OF create-graph--name-ident*])
apply (*auto intro*: *inj-onI*)
done

definition
create-gba φ
$\equiv$ *do* { *nds* $\leftarrow$ *create-graph*$_T$ φ;
RETURN (*create-gba-from-nodes* φ *nds*) }

lemma *create-graph-precond*: *create-graph* φ
$\leq$ *SPEC* (*create-gba-from-nodes-precond* φ)
apply (*clarsimp simp*: *pw-le-iff refine-pw-simps create-graph-nofail*)
apply *unfold-locales*
by *simp*

lemma *create-gba--invar*:
create-gba $\varphi \leq$ *SPEC gba*
unfolding *create-gba-def create-graph-eq-create-graph*$_T$[*symmetric*]
apply (*refine-rcg refine-vcg order-trans*[*OF create-graph-precond*])
by (*rule create-gba-from-nodes-precond.create-gba-from-nodes--invar*)

lemma *create-gba-acc*:
shows *create-gba* $\varphi \leq$ *SPEC*($\lambda\mathcal{A}$. $\forall\, \xi$. *gba.accept* $\mathcal{A}$ $\xi \longleftrightarrow \xi \models_n \varphi$)
unfolding *create-gba-def create-graph-eq-create-graph*$_T$[*symmetric*]
apply (*refine-rcg refine-vcg order-trans*[*OF create-graph-precond*])
using *create-gba-from-nodes-precond.L4-9*
using *create-gba-from-nodes-precond.L4-10*
by *blast*

lemma *create-gba--name-inj*:
shows *create-gba* $\varphi \leq$ *SPEC*($\lambda\mathcal{A}$. (*inj-on name* (*g-V* $\mathcal{A}$)))
unfolding *create-gba-def create-graph-eq-create-graph*$_T$[*symmetric*]

apply (*refine-rcg refine-vcg order-trans*[*OF create-graph--name-inj*])
apply (*auto simp*: *create-gba-from-nodes-def*)
done

lemma *create-gba--fin*:
shows *create-gba* $\varphi \leq SPEC(\lambda\mathcal{A}.\ (finite\ (g\text{-}V\ \mathcal{A})))$
unfolding *create-gba-def create-graph-eq-create-graph*$_T$[*symmetric*]
apply (*refine-rcg refine-vcg order-trans*[*OF create-graph-finite*])
apply (*auto simp*: *create-gba-from-nodes-def*)
done

lemma *create-graph-old-finite*:
create-graph $\varphi \leq SPEC\ (\lambda nds.\ \forall\, nd{\in}nds.\ finite\ (old\ nd))$
proof −
show *?thesis*
unfolding *create-graph-def create-graph-eq-create-graph*$_T$[*symmetric*]
unfolding *expand-def*
apply (*intro refine-vcg*)
apply (*rule-tac order-trans*)
apply (*rule-tac REC-le-RECT*, *refine-mono*)
apply (*fold expand*$_T$*-def*)
apply (*rule-tac order-trans*[*OF expand-term-prop*])
apply *auto*[*1*]
apply (*rule-tac SPEC-rule*)
apply *auto*
by (*metis infinite-super subfrmlsn-finite*)
qed

lemma *create-gba--old-fin*:
shows *create-gba* $\varphi \leq SPEC(\lambda\mathcal{A}.\ \forall\, nd{\in}g\text{-}V\ \mathcal{A}.\ finite\ (old\ nd))$
unfolding *create-gba-def create-graph-eq-create-graph*$_T$[*symmetric*]
apply (*refine-rcg refine-vcg order-trans*[*OF create-graph-old-finite*])
apply (*simp add*: *create-gba-from-nodes-def*)
done

lemma *create-gba--incoming-exists*:
shows *create-gba* φ
$\leq SPEC(\lambda\mathcal{A}.\ \forall\, nd{\in}g\text{-}V\ \mathcal{A}.\ incoming\ nd \subseteq insert\ expand\text{-}init\ (name\ `\ (g\text{-}V\ \mathcal{A})))$
unfolding *create-gba-def create-graph-eq-create-graph*$_T$[*symmetric*]
apply (*refine-rcg refine-vcg order-trans*[*OF create-graph--incoming-name-exist*])
apply (*auto simp add*: *create-gba-from-nodes-def*)
done

lemma *create-gba--no-init*:
shows *create-gba* $\varphi \leq SPEC(\lambda\mathcal{A}.\ expand\text{-}init \notin name\ `\ (g\text{-}V\ \mathcal{A}))$
unfolding *create-gba-def create-graph-eq-create-graph*$_T$[*symmetric*]
apply (*refine-rcg refine-vcg order-trans*[*OF create-graph--incoming-name-exist*])
apply (*auto simp add*: *create-gba-from-nodes-def*)
done

definition *nds-invars nds* ≡
  inj-on name nds
  ∧ *finite nds*
  ∧ *expand-init* ∉ *name‘nds*
  ∧ (∀ *nd*∈*nds*.
    finite (*old nd*)
    ∧ *incoming nd* ⊆ *insert expand-init* (*name ‘ nds*))

lemma *create-gba-nds-invars*: *create-gba* φ ≤ *SPEC* (λ*A*. *nds-invars* (*g-V A*))
  using *create-gba--name-inj*[*of* φ] *create-gba--fin*[*of* φ]
    create-gba--old-fin[*of* φ] *create-gba--incoming-exists*[*of* φ]
    create-gba--no-init[*of* φ]
  unfolding *nds-invars-def*
  apply (*simp add*: *refine-pw-simps pw-le-iff*)
  done

theorem *T4-1*:
  shows *create-gba* φ ≤ *SPEC*(
    λ*A*. *gba A*
    ∧ *finite* (*g-V A*)
    ∧ (∀ ξ. *gba.accept A* ξ ⟷ ξ ⊨*n* φ)
    ∧ (*nds-invars* (*g-V A*)))
  using *create-gba--invar create-gba--fin create-gba-acc create-gba-nds-invars*
  apply (*simp add*: *refine-pw-simps pw-le-iff*)
  apply *blast*
  done

definition *create-name-gba* φ ≡ *do* {
  G ← *create-gba* φ;
  ASSERT (*nds-invars* (*g-V G*));
  RETURN (*gba-rename name G*)
}

theorem *create-name-gba-correct*:
  shows *create-name-gba* φ ≤ *SPEC*(
    λ*A*. *gba A* ∧ *finite* (*g-V A*) ∧ (∀ ξ. *gba.accept A* ξ ⟷ ξ ⊨*n* φ))
  unfolding *create-name-gba-def*
  apply (*refine-rcg refine-vcg order-trans*[*OF T4-1*])
  apply (*simp-all add*: *nds-invars-def gba-rename-correct*)
  done

definition *create-name-igba* :: *'a*::*linorder ltln* ⇒ - **where**
create-name-igba φ ≡ *do* {
  A ← *create-name-gba* φ;
  A' ← *gba-to-idx A*;
  stat-set-data-nres (*card* (*g-V A*)) (*card* (*g-V0 A'*)) (*igbg-num-acc A'*);
  RETURN A'

}

lemma *create-name-igba-correct*: *create-name-igba* $\varphi \leq$ *SPEC* (λG.
  igba $G \wedge$ *finite* (*g-V* G) $\wedge$ ($\forall \xi$. *igba.accept* $G\ \xi \longleftrightarrow \xi \models_n \varphi$))
  unfolding *create-name-igba-def*
  apply (*refine-rcg*
    order-trans[*OF create-name-gba-correct*]
    order-trans[*OF gba.gba-to-idx-ext-correct*]
    refine-vcg)
  apply *clarsimp-all*
proof −
  fix G :: (*nat*, *'a set*) *gba-rec*
  fix A :: *nat set*
  assume *1*: *gba* G
  assume *2*: *finite* (*g-V* G) $A \in$ *gbg-F* G
  interpret *gba* G **using** *1* **by** *this*
  show *finite* A **using** *finite-V-Fe 2* **by** *this*
qed

context
  notes [*refine-vcg*] = *order-trans*[*OF create-name-gba-correct*]
begin
lemma *create-name-igba* $\varphi \leq$ *SPEC* (λG.
  igba $G \wedge$ ($\forall \xi$. *igba.accept* $G\ \xi \longleftrightarrow \xi \models_n \varphi$))
  unfolding *create-name-igba-def*
proof (*refine-rcg refine-vcg*, *clarsimp-all*)
  fix G :: (*nat*, *'a set*) *gba-rec*
  assume *gba* G
  then interpret *gba* G **.**
  note [*refine-vcg*] = *order-trans*[*OF gba-to-idx-ext-correct*]

  assume $\forall \xi$. *gba.accept* $G\ \xi = \xi \models_n \varphi$ *finite* (*g-V* G)
  thus *gba-to-idx* $G \leq$ *SPEC* ($\lambda G'$. *igba* $G' \wedge$ ($\forall \xi$. *igba.accept* $G'\ \xi = \xi \models_n \varphi$))
    by (*refine-rcg refine-vcg*) (*auto intro*: *finite-V-Fe*)
qed

end

end

6 Refinement to Efficient Code

theory *LTL-to-GBA-impl*
imports
  LTL-to-GBA
  ../Datatype-Order-Generator/Order-Generator
  ../CAVA-Automata/Automata-Impl
  ../CAVA-Automata/CAVA-Base/CAVA-Code-Target

begin

6.1 Parametricity Setup Boilerplate

6.1.1 LTL Formulas

derive *linorder ltln*

inductive-set *ltln-rel* **for** R **where**
  $(LTLnTrue,LTLnTrue) \in ltln\text{-}rel\ R$
| $(LTLnFalse,LTLnFalse) \in ltln\text{-}rel\ R$
| $(a,a') \in R \Longrightarrow (LTLnProp\ a,LTLnProp\ a') \in ltln\text{-}rel\ R$
| $(a,a') \in R \Longrightarrow (LTLnNProp\ a,LTLnNProp\ a') \in ltln\text{-}rel\ R$
| $[\![(a,a') \in ltln\text{-}rel\ R;\ (b,b') \in ltln\text{-}rel\ R]\!]$
  $\Longrightarrow (LTLnAnd\ a\ b,LTLnAnd\ a'\ b') \in ltln\text{-}rel\ R$
| $[\![(a,a') \in ltln\text{-}rel\ R;\ (b,b') \in ltln\text{-}rel\ R]\!]$
  $\Longrightarrow (LTLnOr\ a\ b,LTLnOr\ a'\ b') \in ltln\text{-}rel\ R$
| $[\![(a,a') \in ltln\text{-}rel\ R]\!] \Longrightarrow (LTLnNext\ a,LTLnNext\ a') \in ltln\text{-}rel\ R$
| $[\![(a,a') \in ltln\text{-}rel\ R;\ (b,b') \in ltln\text{-}rel\ R]\!]$
  $\Longrightarrow (LTLnUntil\ a\ b,LTLnUntil\ a'\ b') \in ltln\text{-}rel\ R$
| $[\![(a,a') \in ltln\text{-}rel\ R;\ (b,b') \in ltln\text{-}rel\ R]\!]$
  $\Longrightarrow (LTLnRelease\ a\ b,LTLnRelease\ a'\ b') \in ltln\text{-}rel\ R$

lemmas *ltln-rel-induct*[*induct set*]
  = *ltln-rel.induct*[*unfolded relAPP-def*[*of ltln-rel*, *symmetric*]]
lemmas *ltln-rel-cases*[*cases set*]
  = *ltln-rel.cases*[*unfolded relAPP-def*[*of ltln-rel*, *symmetric*]]
lemmas *ltln-rel-intros*
  = *ltln-rel.intros*[*unfolded relAPP-def*[*of ltln-rel*, *symmetric*]]

inductive-simps *ltln-rel-left-simps*[*unfolded relAPP-def*[*of ltln-rel*, *symmetric*]]:
  $(LTLnTrue,z) \in ltln\text{-}rel\ R$
  $(LTLnFalse,z) \in ltln\text{-}rel\ R$
  $(LTLnProp\ p,\ z) \in ltln\text{-}rel\ R$
  $(LTLnNProp\ p,\ z) \in ltln\text{-}rel\ R$
  $(LTLnAnd\ a\ b,\ z) \in ltln\text{-}rel\ R$
  $(LTLnOr\ a\ b,\ z) \in ltln\text{-}rel\ R$
  $(LTLnNext\ a,\ z) \in ltln\text{-}rel\ R$
  $(LTLnUntil\ a\ b,\ z) \in ltln\text{-}rel\ R$
  $(LTLnRelease\ a\ b,\ z) \in ltln\text{-}rel\ R$

lemma *ltln-rel-sv*[*relator-props*]:
  assumes *SV*: *single-valued R*
  shows *single-valued* $(\langle R\rangle ltln\text{-}rel)$
proof (*intro single-valuedI allI impI*)
  fix *x y z*
  assume $(x,\ y) \in \langle R\rangle ltln\text{-}rel\ (x,\ z) \in \langle R\rangle ltln\text{-}rel$
  thus *y*=*z*
    apply (*induction arbitrary*: *z*)

apply (*simp* (*no-asm-use*) *only*: *ltln-rel-left-simps*
| *blast intro*: *single-valuedD*[*OF SV*])+
done
qed

lemma *ltln-rel-id*[*relator-props*]: ⟦ *R* = *Id* ⟧ ⟹ ⟨*R*⟩*ltln-rel* = *Id*
proof (*intro equalityI subsetI*, *clarsimp-all*)
fix *a b*
assume (*a*,*b*)∈⟨*Id*⟩*ltln-rel*
thus *a*=*b*
by *induction auto*
next
fix *a*
show (*a*,*a*)∈⟨*Id*⟩*ltln-rel*
by (*induction a*) (*auto intro*: *ltln-rel-intros*)
qed

lemma *ltln-rel-id-simp*[*simp*]: ⟨*Id*⟩*ltln-rel* = *Id* **by** (*rule ltln-rel-id*) *simp*

consts *i-ltln* :: *interface* ⇒ *interface*
lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of ltln-rel i-ltln*]

thm *ltln-rel-intros*[*no-vars*]

lemma *ltln-con-param*[*param*, *autoref-rules*]:
(*LTLnTrue*, *LTLnTrue*) ∈ ⟨*R*⟩*ltln-rel*
(*LTLnFalse*, *LTLnFalse*) ∈ ⟨*R*⟩*ltln-rel*
(*LTLnProp*, *LTLnProp*) ∈ *R* → ⟨*R*⟩*ltln-rel*
(*LTLnNProp*, *LTLnNProp*) ∈ *R* → ⟨*R*⟩*ltln-rel*
(*LTLnAnd*, *LTLnAnd*) ∈ ⟨*R*⟩*ltln-rel* → ⟨*R*⟩*ltln-rel* → ⟨*R*⟩*ltln-rel*
(*LTLnOr*, *LTLnOr*) ∈ ⟨*R*⟩*ltln-rel* → ⟨*R*⟩*ltln-rel* → ⟨*R*⟩*ltln-rel*
(*LTLnNext*, *LTLnNext*) ∈ ⟨*R*⟩*ltln-rel* → ⟨*R*⟩*ltln-rel*
(*LTLnUntil*, *LTLnUntil*) ∈ ⟨*R*⟩*ltln-rel* → ⟨*R*⟩*ltln-rel* → ⟨*R*⟩*ltln-rel*
(*LTLnRelease*, *LTLnRelease*) ∈ ⟨*R*⟩*ltln-rel* → ⟨*R*⟩*ltln-rel* → ⟨*R*⟩*ltln-rel*
by (*auto intro*: *ltln-rel-intros*)

lemma *case-ltln-param*[*param*, *autoref-rules*]:
(*case-ltln*,*case-ltln*) ∈ *Rv* → *Rv* → (*R* → *Rv*)
→ (*R* → *Rv*)
→ (⟨*R*⟩*ltln-rel* → ⟨*R*⟩*ltln-rel* → *Rv*)
→ (⟨*R*⟩*ltln-rel* → ⟨*R*⟩*ltln-rel* → *Rv*)
→ (⟨*R*⟩*ltln-rel* → *Rv*)
→ (⟨*R*⟩*ltln-rel* → ⟨*R*⟩*ltln-rel* → *Rv*)
→ (⟨*R*⟩*ltln-rel* → ⟨*R*⟩*ltln-rel* → *Rv*) → ⟨*R*⟩*ltln-rel* → *Rv*
apply (*clarsimp*)
apply (*case-tac ai*, *simp-all add*: *ltln-rel-left-simps*)
apply (*clarsimp-all*)
apply *parametricity*+

done

lemma *rec-ltln-param*[*param*, *autoref-rules*]:
(*rec-ltln*,*rec-ltln*) ∈ *Rv* → *Rv* → (*R* → *Rv*)
→ (*R* → *Rv*)
→ (⟨*R*⟩*ltln-rel* → ⟨*R*⟩*ltln-rel* → *Rv* → *Rv* → *Rv*)
→ (⟨*R*⟩*ltln-rel* → ⟨*R*⟩*ltln-rel* → *Rv* → *Rv* → *Rv*)
→ (⟨*R*⟩*ltln-rel* → *Rv* → *Rv*)
→ (⟨*R*⟩*ltln-rel* → ⟨*R*⟩*ltln-rel* → *Rv* → *Rv* → *Rv*)
→ (⟨*R*⟩*ltln-rel* → ⟨*R*⟩*ltln-rel* → *Rv* → *Rv* → *Rv*)
→ ⟨*R*⟩*ltln-rel* → *Rv*
proof *clarsimp*
case *goal1*
from *goal1*(*10*)
show *?case*
apply (*induction*)
using *goal1*(*1*−*9*)
apply *simp-all*
apply *parametricity*+
done
qed

lemma *case-ltln-mono*[*refine-mono*]:
assumes φ = *LTLnTrue* ⟹ *a*≤*a*′
assumes φ = *LTLnFalse* ⟹ *b*≤*b*′
assumes ⋀*p*. φ = *LTLnProp p* ⟹ *c p*≤*c*′ *p*
assumes ⋀*p*. φ = *LTLnNProp p* ⟹ *d p*≤*d*′ *p*
assumes ⋀μ ν. φ = *LTLnAnd* μ ν ⟹ *e* μ ν≤*e*′ μ ν
assumes ⋀μ ν. φ = *LTLnOr* μ ν ⟹ *f* μ ν≤*f*′ μ ν
assumes ⋀μ. φ = *LTLnNext* μ ⟹ *g* μ≤*g*′ μ
assumes ⋀μ ν. φ = *LTLnUntil* μ ν ⟹ *h* μ ν≤*h*′ μ ν
assumes ⋀μ ν. φ = *LTLnRelease* μ ν ⟹ *i* μ ν≤*i*′ μ ν
shows *case-ltln a b c d e f g h i* φ ≤ *case-ltln a*′ *b*′ *c*′ *d*′ *e*′ *f*′ *g*′ *h*′ *i*′ φ
using *assms*
apply (*cases* φ)
apply *simp-all*
done

primrec *ltln-eq* **where**
ltln-eq eq LTLnTrue f ⟷ (*case f of LTLnTrue* ⇒ *True* | - ⇒ *False*)
| *ltln-eq eq LTLnFalse f* ⟷ (*case f of LTLnFalse* ⇒ *True* | - ⇒ *False*)
| *ltln-eq eq* (*LTLnProp p*) *f* ⟷ (*case f of LTLnProp p*′ ⇒ *eq p p*′ | - ⇒ *False*)
| *ltln-eq eq* (*LTLnNProp p*) *f* ⟷ (*case f of LTLnNProp p*′ ⇒ *eq p p*′ | - ⇒ *False*)
| *ltln-eq eq* (*LTLnAnd* μ ν) *f*
⟷ (*case f of LTLnAnd* μ′ ν′ ⇒ *ltln-eq eq* μ μ′ ∧ *ltln-eq eq* ν ν′ | - ⇒ *False*)
| *ltln-eq eq* (*LTLnOr* μ ν) *f*
⟷ (*case f of LTLnOr* μ′ ν′ ⇒ *ltln-eq eq* μ μ′ ∧ *ltln-eq eq* ν ν′ | - ⇒ *False*)
| *ltln-eq eq* (*LTLnNext* φ) *f*

$\longleftrightarrow$ (*case f of LTLnNext* φ' $\Rightarrow$ *ltln-eq eq* φ φ' | - $\Rightarrow$ *False*)
| *ltln-eq eq* (*LTLnUntil* μ ν) *f*
$\longleftrightarrow$ (*case f of LTLnUntil* μ' ν' $\Rightarrow$ *ltln-eq eq* μ $\mu' \wedge$ *ltln-eq eq* ν ν' | - $\Rightarrow$ *False*)
| *ltln-eq eq* (*LTLnRelease* μ ν) *f*
$\longleftrightarrow$ (*case f of*
LTLnRelease μ' ν' $\Rightarrow$ *ltln-eq eq* μ $\mu' \wedge$ *ltln-eq eq* ν ν'
| - $\Rightarrow$ *False*)

lemma *ltln-eq-autoref*[*autoref-rules*]:
assumes *EQP*: (*eq*,*op*=) $\in R \to R \to$ *bool-rel*
shows (*ltln-eq eq*, *op*=) $\in \langle R \rangle$*ltln-rel* $\to \langle R \rangle$*ltln-rel* $\to$ *bool-rel*
proof (*intro fun-relI*)
fix μ' μ ν' ν
assume (μ',μ)$\in\langle R\rangle$*ltln-rel* **and** (ν',ν)$\in\langle R\rangle$*ltln-rel*
thus (*ltln-eq eq* μ' ν', μ=ν)$\in$*bool-rel*
apply (*induction arbitrary*: ν' ν)
apply (*erule ltln-rel-cases*, *simp-all*) []
apply (*erule ltln-rel-cases*, *simp-all*) []
apply (*erule ltln-rel-cases*,
simp-all add: *EQP*[*THEN fun-relD*, *THEN fun-relD*, *THEN IdD*]) []
apply (*erule ltln-rel-cases*,
simp-all add: *EQP*[*THEN fun-relD*, *THEN fun-relD*, *THEN IdD*]) []
apply (*rotate-tac* -1)
apply (*erule ltln-rel-cases*, *simp-all*) []
apply (*rotate-tac* -1)
apply (*erule ltln-rel-cases*, *simp-all*) []
apply (*rotate-tac* -1)
apply (*erule ltln-rel-cases*, *simp-all*) []
apply (*rotate-tac* -1)
apply (*erule ltln-rel-cases*, *simp-all*) []
apply (*rotate-tac* -1)
apply (*erule ltln-rel-cases*, *simp-all*) []
done
qed

lemma *ltln-dflt-cmp*[*autoref-rules-raw*]:
assumes *PREFER-id R*
shows
(*dflt-cmp op* $\leq$ *op* $<$, *dflt-cmp op* $\leq$ *op* $<$)
$\in \langle R \rangle$*ltln-rel* $\to \langle R \rangle$*ltln-rel* $\to$ *comp-res-rel*
using *assms*
by *simp*

type-synonym
node-name-impl = *node-name*

abbreviation (*input*) *node-name-rel* ≡ *Id* :: (*node-name-impl*×*node-name*) *set*

lemma *case-ltln-gtransfer*:
assumes
$\gamma\ ai \leq a$
$\gamma\ bi \leq b$
$\bigwedge a.\ \gamma\ (ci\ a) \leq c\ a$
$\bigwedge a.\ \gamma\ (di\ a) \leq d\ a$
$\bigwedge ltln1\ ltln2.\ \gamma\ (ei\ ltln1\ ltln2) \leq e\ ltln1\ ltln2$
$\bigwedge ltln1\ ltln2.\ \gamma\ (fi\ ltln1\ ltln2) \leq f\ ltln1\ ltln2$
$\bigwedge ltln.\ \gamma\ (gi\ ltln) \leq g\ ltln$
$\bigwedge ltln1\ ltln2.\ \gamma\ (hi\ ltln1\ ltln2) \leq h\ ltln1\ ltln2$
$\bigwedge ltln1\ ltln2.\ \gamma\ (ii\ ltln1\ ltln2) \leq i\ ltln1\ ltln2$
shows $\gamma\ (case\text{-}ltln\ ai\ bi\ ci\ di\ ei\ fi\ gi\ hi\ ii\ \varphi)$
$\leq (case\text{-}ltln\ a\ b\ c\ d\ e\ f\ g\ h\ i\ \varphi)$
apply (*cases* φ)
apply (*auto intro*: *assms*)
done

lemmas [*refine-transfer*]
= *case-ltln-gtransfer*[**where** γ=*nres-of*] *case-ltln-gtransfer*[**where** γ=*RETURN*]

lemma [*refine-transfer*]:
assumes
$ai \neq dSUCCEED$
$bi \neq dSUCCEED$
$\bigwedge a.\ ci\ a \neq dSUCCEED$
$\bigwedge a.\ di\ a \neq dSUCCEED$
$\bigwedge ltln1\ ltln2.\ ei\ ltln1\ ltln2 \neq dSUCCEED$
$\bigwedge ltln1\ ltln2.\ fi\ ltln1\ ltln2 \neq dSUCCEED$
$\bigwedge ltln.\ gi\ ltln \neq dSUCCEED$
$\bigwedge ltln1\ ltln2.\ hi\ ltln1\ ltln2 \neq dSUCCEED$
$\bigwedge ltln1\ ltln2.\ ii\ ltln1\ ltln2 \neq dSUCCEED$
shows $case\text{-}ltln\ ai\ bi\ ci\ di\ ei\ fi\ gi\ hi\ ii\ \varphi \neq dSUCCEED$
apply (*cases* φ)
apply (*simp-all add*: *assms*)
done

6.1.2 Nodes

record $'a$ *node-impl* =
name-impl :: *node-name-impl*
incoming-impl :: (*node-name-impl*,*unit*) *RBT-Impl.rbt*
new-impl :: $'a$ *frml list*
old-impl :: $'a$ *frml list*
next-impl :: $'a$ *frml list*

definition *node-rel* **where** *node-rel-def-internal*: *node-rel Re R* ≡ {(

(| *name-impl* = *namei*,
incoming-impl = *inci*,
new-impl = *newi*,
old-impl = *oldi*,
next-impl = *nexti*,
... = *morei*
|),
(| *name* = *name*,
incoming = *inc*,
new=*new*,
old=*old*,
next = *next*,
... = *more*
|) | *namei name inci inc newi new oldi old nexti next morei more*.
(*namei*,*name*)∈*node-name-rel*
∧ (*inci*,*inc*)∈⟨*node-name-rel*⟩*dflt-rs-rel*
∧ (*newi*,*new*)∈⟨⟨*R*⟩*ltln-rel*⟩*lss.rel*
∧ (*oldi*,*old*)∈⟨⟨*R*⟩*ltln-rel*⟩*lss.rel*
∧ (*nexti*,*next*)∈⟨⟨*R*⟩*ltln-rel*⟩*lss.rel*
∧ (*morei*,*more*)∈*Re*
}

lemma *node-rel-def*: ⟨*Re*,*R*⟩*node-rel* = {(
(| *name-impl* = *namei*,
incoming-impl = *inci*,
new-impl = *newi*,
old-impl = *oldi*,
next-impl = *nexti*,
... = *morei*
|),
(| *name* = *name*,
incoming = *inc*,
new=*new*,
old=*old*,
next = *next*,
... = *more*
|) | *namei name inci inc newi new oldi old nexti next morei more*.
(*namei*,*name*)∈*node-name-rel*
∧ (*inci*,*inc*)∈⟨*node-name-rel*⟩*dflt-rs-rel*
∧ (*newi*,*new*)∈⟨⟨*R*⟩*ltln-rel*⟩*lss.rel*
∧ (*oldi*,*old*)∈⟨⟨*R*⟩*ltln-rel*⟩*lss.rel*
∧ (*nexti*,*next*)∈⟨⟨*R*⟩*ltln-rel*⟩*lss.rel*
∧ (*morei*,*more*)∈*Re*
} **by** (*simp add*: *node-rel-def-internal relAPP-def*)

lemma *node-rel-sv*[*relator-props*]:
single-valued Re ⟹ *single-valued R* ⟹ *single-valued* (⟨*Re*,*R*⟩*node-rel*)
apply (*rule single-valuedI*)

apply (*simp add*: *node-rel-def*)
apply (*auto*
dest: *single-valuedD lss.rel-sv*[*OF ltln-rel-sv*] *map2set-rel-sv*[*OF ahm-rel-sv*]
dest: *single-valuedD*[
OF map2set-rel-sv[*OF rbt-map-rel-sv*[*OF single-valued-Id single-valued-Id*]]
])
done

consts *i-node* :: *interface* ⇒ *interface* ⇒ *interface*

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of node-rel i-node*]

lemma [*autoref-rules*]: (*node-impl-ext*, *node-ext*) ∈
node-name-rel
→ ⟨*node-name-rel*⟩*dflt-rs-rel*
→ ⟨⟨*R*⟩*ltln-rel*⟩*lss.rel*
→ ⟨⟨*R*⟩*ltln-rel*⟩*lss.rel*
→ ⟨⟨*R*⟩*ltln-rel*⟩*lss.rel*
→ *Re*
→ ⟨*Re*,*R*⟩*node-rel*
unfolding *node-rel-def*
by *auto*

term *node.incoming-update*

lemma [*autoref-rules*]:
(*node-impl.name-impl-update*,*node.name-update*)
∈ (*node-name-rel* → *node-name-rel*) → ⟨*Re*,*R*⟩*node-rel* → ⟨*Re*,*R*⟩*node-rel*
(*node-impl.incoming-impl-update*,*node.incoming-update*)
∈ (⟨*node-name-rel*⟩*dflt-rs-rel* → ⟨*node-name-rel*⟩*dflt-rs-rel*)
→ ⟨*Re*,*R*⟩*node-rel*
→ ⟨*Re*,*R*⟩*node-rel*
(*node-impl.new-impl-update*,*node.new-update*)
∈ (⟨⟨*R*⟩*ltln-rel*⟩*lss.rel* → ⟨⟨*R*⟩*ltln-rel*⟩*lss.rel*) → ⟨*Re*,*R*⟩*node-rel* → ⟨*Re*,*R*⟩*node-rel*
(*node-impl.old-impl-update*,*node.old-update*)
∈ (⟨⟨*R*⟩*ltln-rel*⟩*lss.rel* → ⟨⟨*R*⟩*ltln-rel*⟩*lss.rel*) → ⟨*Re*,*R*⟩*node-rel* → ⟨*Re*,*R*⟩*node-rel*
(*node-impl.next-impl-update*,*node.next-update*)
∈ (⟨⟨*R*⟩*ltln-rel*⟩*lss.rel* → ⟨⟨*R*⟩*ltln-rel*⟩*lss.rel*) → ⟨*Re*,*R*⟩*node-rel* → ⟨*Re*,*R*⟩*node-rel*
(*node-impl.more-update*,*node.more-update*)
∈ (*Re* → *Re*) → ⟨*Re*,*R*⟩*node-rel* → ⟨*Re*,*R*⟩*node-rel*
unfolding *node-rel-def*
by (*auto dest*: *fun-relD*)

term *name*
lemma [*autoref-rules*]:
(*node-impl.name-impl*,*node.name*)∈⟨*Re*,*R*⟩*node-rel* → *node-name-rel*
(*node-impl.incoming-impl*,*node.incoming*)
∈ ⟨*Re*,*R*⟩*node-rel* → ⟨*node-name-rel*⟩*dflt-rs-rel*

```
(node-impl.new-impl,node.new)∈⟨Re,R⟩node-rel → ⟨⟨R⟩ltln-rel⟩lss.rel
(node-impl.old-impl,node.old)∈⟨Re,R⟩node-rel → ⟨⟨R⟩ltln-rel⟩lss.rel
(node-impl.next-impl,node.next)∈⟨Re,R⟩node-rel → ⟨⟨R⟩ltln-rel⟩lss.rel
(node-impl.more, node.more)∈⟨Re,R⟩node-rel → Re
unfolding node-rel-def by auto
```

6.2 Massaging the Abstract Algorithm

In a first step, we do some refinement steps on the abstract data structures, with the goal to make the algorithm more efficient.

6.2.1 Creation of the Nodes

In the expand-algorithm, we replace nested conditionals by case-distinctions, and slightly stratify the code.

```
abbreviation (input) expand2 exp n ns φ n1 nx1 n2 ≡ do {
  (nm, nds) ← exp (
    n(|
      new := insert n1 (new n),
      old := insert φ (old n),
      next := nx1 ∪ next n |),
    ns);
  exp (n(| name := nm, new := n2 ∪ new n, old := {φ} ∪ old n |), nds)
}

context begin interpretation LTL-Syntax .

definition expand-aimpl ≡ REC_T (λexpand (n,ns).
    if new n = {} then (
      if (∃n'∈ns. old n' = old n ∧ next n' = next n) then
        RETURN (name n, upd-incoming n ns)
      else do {
        ASSERT (n ∉ ns);
        ASSERT (name n ∉ name`ns);
        expand (|
          name=expand-new-name (name n),
          incoming={name n},
          new=next n,
          old={},
          next={} |),
          insert n ns)
      }
    ) else do {
      φ ← SPEC (λx. x∈(new n));
      let n = n(| new := new n - {φ} |);
      case φ of
        prop_n(q) ⇒
```

```
        if nprop_n(q)∈old n then RETURN (name n, ns)
        else expand (n(| old := {φ} ∪ old n |), ns)
    | nprop_n(q) ⇒
        if prop_n(q)∈old n then RETURN (name n, ns)
        else expand (n(| old := {φ} ∪ old n |), ns)
    | true_n ⇒ expand (n(| old := {φ} ∪ old n |), ns)
    | false_n ⇒ RETURN (name n, ns)
    | ν and_n μ ⇒ expand (n(|
        new := insert ν (insert μ (new n)),
        old := {φ} ∪ old n,
        next := next n |), ns)
    | X_n ν ⇒ expand
        (n(| new := new n, old := {φ} ∪ old n, next := insert ν (next n) |), ns)
    | μ or_n ν ⇒ expand2 expand n ns φ μ {} {ν}
    | μ U_n ν ⇒ expand2 expand n ns φ μ {φ} {ν}
    | μ V_n ν ⇒ expand2 expand n ns φ ν {φ} {μ,ν}
    (*| - ⇒ do {
      (nm, nds) ← expand (
        n(|
          new := new1 φ ∪ new n,
          old := {φ} ∪ old n,
          next := next1 φ ∪ next n |),
        ns);
      expand (n(| name := nm, new := new2 φ ∪ new n, old := {φ} ∪ old n |),
nds)
    }*)
  }
 )

end

lemma expand-aimpl-refine:
  fixes n-ns :: ('a node × -)
  defines R ≡ Id ∩ {(-,(n,ns)). ∀ n'∈ns. n > name n'}
  defines R' ≡ Id ∩ {(-,(n,ns)). ∀ n'∈ns. name n > name n'}
  assumes [refine]: (n-ns',n-ns)∈R'
  shows expand-aimpl n-ns' ≤ ⇓R (expand_T n-ns)
  using [[goals-limit = 1]]
proof −
  have [relator-props]: single-valued R
    by (auto simp add: R-def intro: single-valuedI)
  have [relator-props]: single-valued R'
    by (auto simp add: R'-def intro: single-valuedI)

  {
    fix n :: 'a node and ns and n' ns'
    assume ((n', ns'), (n, ns)) ∈ R'
    hence (RETURN (name n', ns') ≤ ⇓ R (RETURN (name n, ns)))
      by (auto simp: R-def R'-def pw-le-iff refine-pw-simps)
```

```
  } note aux = this


  show ?thesis
    unfolding expand-aimpl-def expand_T-def
    apply refine-rcg
    apply (simp add: R-def R'-def)
    apply (simp add: R-def R'-def)
    apply (auto simp add: R-def R'-def upd-incoming-def) []
    apply (auto simp add: R-def R'-def upd-incoming-def) []
    apply (auto simp add: R-def R'-def upd-incoming-def) []
    apply rprems
    apply (auto simp: R'-def expand-new-name-def) []
    apply (simp add: R'-def)

    apply (auto split: ltln.split) []
    apply (fastforce simp: R'-def) []
    apply (refine-rcg aux)
    apply (refine-rcg aux)
    apply (auto simp: R'-def) []
    apply (auto simp: R'-def) []

    apply (fastforce simp: R'-def) []
    apply (refine-rcg aux)
    apply (fastforce simp: R'-def) []
    apply (fastforce simp: R'-def) []
    apply (fastforce simp: R'-def) []
    apply (fastforce simp: R'-def) []
    apply (refine-rcg, rprems, (fastforce simp: R-def R'-def)+) []
    apply (fastforce simp: R'-def) []
    apply (refine-rcg, rprems, (fastforce simp: R-def R'-def)+) []
    apply (refine-rcg, rprems, (fastforce simp: R-def R'-def)+) []
    done
qed


thm create-graph-def
definition create-graph-aimpl φ = do {
  (-, nds) ←
  expand-aimpl
   (⦇name = expand-new-name expand-init, incoming = {expand-init},
      new = {φ}, old = {}, next = {}⦈,
   {});
  RETURN nds
}

lemma create-graph-aimpl-refine: create-graph-aimpl φ ≤ ⇓Id (create-graph_T φ)
  unfolding create-graph-aimpl-def create-graph_T-def
  apply (refine-rcg expand-aimpl-refine)
```

apply *auto*
done

6.2.2 Creation of GBA from Nodes

We summarize creation of the GBA and renaming of the nodes into one step

lemma *create-name-gba-alt*: *create-name-gba* φ = *do* {
nds ← *create-graph*$_T$ φ;
ASSERT (*nds-invars nds*);
RETURN (*gba-rename-ext* (λ-. ()) *name* (*create-gba-from-nodes* φ *nds*))
}
proof −
have [*simp*]: ⋀*nds*. *g-V* (*create-gba-from-nodes* φ *nds*) = *nds*
by (*auto simp*: *create-gba-from-nodes-def*)

show *?thesis*
unfolding *create-name-gba-def create-gba-def*
by *simp*
qed

In the following, we implement the componenents of the renamed GBA separately.

definition *build-succ nds* =
FOREACH
nds (λ*q*′ *s*.
FOREACH
(*incoming q*′) (λ*qn s*.
if qn=*expand-init then*
RETURN s
else
RETURN (*s*(*qn* ↦ *insert* (*name q*′) (*the-default* {} (*s qn*))))
) *s*
) *Map.empty*

lemma *build-succ-aux1*:
assumes [*simp*]: *finite nds*
assumes [*simp*]: ⋀*q*. *q*∈*nds* ⟹ *finite* (*incoming q*)
shows *build-succ nds* ≤ *SPEC* (λ*r*. *r* = (λ*qn*.
dflt-None-set {*qn*′. ∃ *q*′.
q′∈*nds* ∧ *qn*′ = *name q*′ ∧ *qn*∈*incoming q*′ ∧ *qn*≠*expand-init*
}))
unfolding *build-succ-def*
apply (*refine-rcg refine-vcg*
FOREACH-rule[**where**
I=λ*it s*. *s* = (λ*qn*. *dflt-None-set* {*qn*′. ∃ *q*′. *q*′∈*nds*−*it* ∧ *qn*′ = *name q*′
∧ *qn*∈*incoming q*′ ∧ *qn*≠*expand-init* })])

```
  apply (simp-all add: dflt-None-set-def) [2]
  apply (rename-tac q' it s)
  apply (rule-tac I=λit2 s. s =
    (λqn. dflt-None-set (
      {qn'. ∃ q'. q'∈nds−it ∧ qn' = name q' ∧ qn∈incoming q' ∧ qn≠expand-init
} ∪
      {qn' . qn'=name q' ∧ qn∈incoming q' − it2 ∧ qn≠expand-init} ))
    in FOREACH-rule)

  apply auto []

  apply (simp-all add: dflt-None-set-def)

  apply (refine-rcg refine-vcg)
  apply (auto simp: dflt-None-set-def intro!: ext) []
  apply (rule ext, (auto) [])+
  done

lemma build-succ-aux2:
  assumes NINIT: expand-init ∉ name‘nds
  assumes CL: ⋀nd. nd∈nds ⟹ incoming nd ⊆ insert expand-init (name‘nds)
  shows
  (λqn. dflt-None-set
    {qn'. ∃ q'. q'∈nds ∧ qn' = name q' ∧ qn∈incoming q' ∧ qn≠expand-init })
  = (λqn. dflt-None-set (succ-of-E
       (rename-E name {(q, q'). q ∈ nds ∧ q' ∈ nds ∧ name q ∈ incoming q'})
qn))
  (is (λqn. dflt-None-set (?L qn)) = (λqn. dflt-None-set (?R qn)))
  apply (intro ext)
  apply (fo-rule arg-cong)
proof (intro ext equalityI subsetI)
  fix qn x
  assume x∈?R qn
  thus x∈?L qn using NINIT
    by (force simp: succ-of-E-def)
next
  fix qn x
  assume XL: x∈?L qn
  show x∈?R qn
  proof (cases qn = expand-init)
    case False
    from XL obtain q' where
      A: q'∈nds qn∈incoming q'
      and [simp]: x=name q'
      by auto
    from False obtain q where B: q∈nds and [simp]: qn = name q
      using CL A by auto

    from A B show x∈?R qn
```

by (*auto simp*: *succ-of-E-def image-def*)
next
case *True*[*simp*]
from *XL* **show** *x*∈*?R qn* **by** *simp*
qed
qed

lemma *build-succ-correct*:
assumes *NINIT*: *expand-init* ∉ *name*‘*nds*
assumes *FIN*: *finite nds*
assumes *CL*: ⋀*nd*. *nd*∈*nds* ⟹ *incoming nd* ⊆ *insert expand-init* (*name*‘*nds*)
shows *build-succ nds* ≤ *SPEC* (λ*r*.
E-of-succ (λ*qn*. *the-default* {} (*r qn*))
= *rename-E* (λ*u*. *name u*) {(*q*, *q*′). *q* ∈ *nds* ∧ *q*′ ∈ *nds* ∧ *name q* ∈ *incoming* *q*′})
proof −
from *FIN CL* **have** *FIN*′: ⋀*q*. *q*∈*nds* ⟹ *finite* (*incoming q*)
by (*metis finite-imageI finite-insert infinite-super*)

note *build-succ-aux1*[*OF FIN FIN*′]
also note *build-succ-aux2*[*OF NINIT CL*]
finally show *?thesis*
by (*rule order-trans*) *auto*
qed

context begin interpretation *LTL-Syntax* .
primrec *until-frmlsn* :: ′*a frml* ⇒ (′*a frml* × ′*a frml*) *set* **where**
until-frmlsn (*μ and*$_n$ *ψ*) = (*until-frmlsn μ*) ∪ (*until-frmlsn ψ*)
| *until-frmlsn* (*X*$_n$ *μ*) = *until-frmlsn μ*
| *until-frmlsn* (*μ U*$_n$ *ψ*) = *insert* (*μ*, *ψ*) ((*until-frmlsn μ*) ∪ (*until-frmlsn ψ*))
| *until-frmlsn* (*μ V*$_n$ *ψ*) = (*until-frmlsn μ*) ∪ (*until-frmlsn ψ*)
| *until-frmlsn* (*μ or*$_n$ *ψ*) = (*until-frmlsn μ*) ∪ (*until-frmlsn ψ*)
| *until-frmlsn* (*true*$_n$) = {}
| *until-frmlsn* (*false*$_n$) = {}
| *until-frmlsn* (*prop*$_n$(-)) = {}
| *until-frmlsn* (*nprop*$_n$(-)) = {}

end

lemma *until-frmlsn-correct*:
until-frmlsn φ = {(*μ*, *η*). *LTLnUntil μ η* ∈ *subfrmlsn φ*}
by (*induct φ*) *auto*

definition *build-F nds φ*
≡ (λ(*μ*,*η*). *name* ‘ {*q* ∈ *nds*. (*LTLnUntil μ η* ∈ *old q* ⟶ *η* ∈ *old q*)}) ‘
until-frmlsn φ

lemma *build-F-correct*: *build-F nds* φ =
{*name* ‘ *A* |*A*. ∃μ η. *A* = {*q* ∈ *nds*. *LTLnUntil* μ η ∈ *old q* ⟶ η ∈ *old q*} ∧
LTLnUntil μ η ∈ *subfrmlsn* φ}
proof −
have {*name* ‘ *A* |*A*. ∃μ η. *A* = {*q* ∈ *nds*. *LTLnUntil* μ η ∈ *old q* ⟶ η ∈ *old q*} ∧
LTLnUntil μ η ∈ *subfrmlsn* φ}
= (λ(μ,η). *name*‘{*q*∈*nds*. *LTLnUntil* μ η ∈ *old q* ⟶ η ∈ *old q*})
‘ {(μ, η). *LTLnUntil* μ η ∈ *subfrmlsn* φ}
by *auto*
also have ... = (λ(μ,η). *name*‘{*q*∈*nds*. *LTLnUntil* μ η ∈ *old q* ⟶ η ∈ *old q*})
‘ *until-frmlsn* φ
unfolding *until-frmlsn-correct* **..**
finally show *?thesis*
unfolding *build-F-def* **by** *simp*
qed

definition *pn-props ps* ≡ *FOREACHi*
(λ*it* (*P*,*N*). *P* = {*p*. *LTLnProp p* ∈ *ps* − *it*} ∧ *N* = {*p*. *LTLnNProp p* ∈ *ps* − *it*})
ps (λ*p* (*P*,*N*).
case p of LTLnProp p ⇒ *RETURN* (*insert p P*,*N*)
| *LTLnNProp p* ⇒ *RETURN* (*P*, *insert p N*)
| - ⇒ *RETURN* (*P*,*N*)
) ({},{})

lemma *pn-props-correct*:
assumes [*simp*]: *finite ps*
shows *pn-props ps* ≤ *SPEC*(λ*r*. *r* =
({*p*. *LTLnProp p* ∈ *ps*}, {*p*. *LTLnNProp p* ∈ *ps*}))
unfolding *pn-props-def*
apply (*refine-rcg refine-vcg*)
apply (*auto split*: *ltln.split*)
done

definition *pn-map nds* ≡ *FOREACH nds*
(λ*nd m*. *do* {
PN ← *pn-props* (*old nd*);
RETURN (*m*(*name nd* ↦ *PN*))
}) *Map.empty*

lemma *pn-map-correct*:
assumes [*simp*]: *finite nds*
assumes *FIN'*: ⋀*nd*. *nd*∈*nds* ⟹ *finite* (*old nd*)
assumes *INJ*: *inj-on name nds*
shows *pn-map nds* ≤ *SPEC* (λ*r*. ∀ *qn*.

```
    case r qn of
      None ⇒ qn ∉ name‘nds
    | Some (P,N) ⇒ qn ∈ name‘nds
      ∧ P = {p. LTLnProp p ∈ old (the-inv-into nds name qn)}
      ∧ N = {p. LTLnNProp p ∈ old (the-inv-into nds name qn)}
  )
  unfolding pn-map-def
  apply (refine-rcg refine-vcg
    FOREACH-rule[where I=λit r. ∀ qn.
      case r qn of
        None ⇒ qn ∉ name‘(nds − it)
      | Some (P,N) ⇒ qn ∈ name‘(nds − it)
        ∧ P = {p. LTLnProp p ∈ old (the-inv-into nds name qn)}
        ∧ N = {p. LTLnNProp p ∈ old (the-inv-into nds name qn)}]
    order-trans[OF pn-props-correct]
  )
  apply simp-all
  apply (blast dest: set-mp[THEN FIN'']) []
  apply (force
    split: option.splits
    simp: the-inv-into-f-f[OF INJ] it-step-insert-iff) []
  apply (fastforce split: option.splits) []
  done



definition cr-rename-gba nds φ ≡ do {
  let V = name ‘ nds;
  let V0 = name ‘ {q ∈ nds. expand-init ∈ incoming q};
  succmap ← build-succ nds;
  let E = E-of-succ (the-default {} o succmap);
  let F = build-F nds φ;
  pnm ← pn-map nds;
  let L = (λqn l. case pnm qn of
      None ⇒ False
    | Some (P,N) ⇒ (∀ p∈P. p∈(l:::r⟨Id⟩fun-set-rel)) ∧ (∀ p∈N. p∉l)
  );
  RETURN (⦇ g-V = V, g-E=E, g-V0=V0, gbg-F = F, gba-L = L ⦈)
}

lemma cr-rename-gba-refine:
  assumes INV: nds-invars nds
  assumes REL[simplified]: (nds',nds)∈Id (φ',φ)∈Id
  shows cr-rename-gba nds' φ'
  ≤ ⇓Id (RETURN (gba-rename-ext (λ-. ()) name (create-gba-from-nodes φ nds)))
  unfolding RETURN-SPEC-conv
proof (rule Id-SPEC-refine)
  from INV have
    NINIT: expand-init ∉ name‘nds
```

```
    and FIN: finite nds
    and FIN': ⋀nd. nd∈nds ⟹ finite (old nd)
    and CL: ⋀nd. nd∈nds ⟹ incoming nd ⊆ insert expand-init (name'nds)
    and INJ: inj-on name nds
    unfolding nds-invars-def by auto
  show cr-rename-gba nds' φ'
   ≤ SPEC (λx. x = gba-rename-ext (λ-. ()) name (create-gba-from-nodes φ nds))
    unfolding REL
    unfolding cr-rename-gba-def
    apply (refine-rcg refine-vcg
      order-trans[OF build-succ-correct[OF NINIT FIN CL]]
      order-trans[OF pn-map-correct[OF FIN FIN' INJ]]
    )
    unfolding gba-rename-ecnv-def gb-rename-ecnv-def
      fr-rename-ext-def create-gba-from-nodes-def
    apply simp
    apply (intro conjI)
    apply (simp add: comp-def)
    apply (simp add: build-F-correct)
    apply (intro ext)
    apply (drule-tac x=qn in spec)
    apply (auto simp: the-inv-into-f-f[OF INJ] split: option.split prod.split)
    done
qed

definition create-name-gba-aimpl φ ≡ do {
  nds ← create-graph-aimpl φ;
  ASSERT (nds-invars nds);
  cr-rename-gba nds φ
}


lemma create-name-gba-aimpl-refine:
  create-name-gba-aimpl φ ≤ ⇓Id (create-name-gba φ)
  unfolding create-name-gba-aimpl-def create-name-gba-alt
  apply (refine-rcg create-graph-aimpl-refine cr-rename-gba-refine)
  by auto
```

6.3 Refinement to Efficient Data Structures

6.3.1 Creation of GBA from Nodes

```
schematic-lemma until-frmlsn-impl-aux:
  assumes [relator-props, simp]: R=Id
  shows (?c,until-frmlsn)
  ∈ ⟨(R::(-×-::linorder) set)⟩ltln-rel → ⟨⟨R⟩ltln-rel ×r ⟨R⟩ltln-rel⟩dflt-rs-rel
  unfolding until-frmlsn-def
  apply (autoref (keep-goal, trace))
  done
```

concrete-definition *until-frmlsn-impl* **uses** *until-frmlsn-impl-aux*
lemmas [*autoref-rules*] = *until-frmlsn-impl.refine*[*OF PREFER-id-D*]

schematic-lemma *build-succ-impl-aux*:
shows (*?c*,*build-succ*) ∈
⟨⟨*Rm*,*R*⟩*node-rel*⟩*list-set-rel*
→ ⟨⟨*nat-rel*,⟨*nat-rel*⟩*list-set-rel*⟩*iam-map-rel*⟩*nres-rel*
unfolding *build-succ-def*[*abs-def*] *expand-init-def*
using [[*autoref-trace-failed-id*]]
apply (*autoref* (*keep-goal*, *trace*))
done

concrete-definition *build-succ-impl* **uses** *build-succ-impl-aux*
lemmas [*autoref-rules*] = *build-succ-impl.refine*

schematic-lemma *build-succ-code-aux*: *RETURN ?c* ≤ *build-succ-impl x*
unfolding *build-succ-impl-def*
apply (*refine-transfer* (*post*))
done

concrete-definition *build-succ-code* **uses** *build-succ-code-aux*
lemmas [*refine-transfer*] = *build-succ-code.refine*

schematic-lemma *build-F-impl-aux*:
assumes [*relator-props*]: *R* = *Id*
shows (*?c*,*build-F*) ∈
⟨⟨*Rm*,*R*⟩*node-rel*⟩*list-set-rel* → ⟨*R*⟩*ltln-rel* → ⟨⟨*nat-rel*⟩*list-set-rel*⟩*list-set-rel*
unfolding *build-F-def*[*abs-def*]
using [[*autoref-trace-failed-id*]]
apply (*autoref* (*keep-goal*, *trace*))
done

concrete-definition *build-F-impl* **uses** *build-F-impl-aux*
lemmas [*autoref-rules*] = *build-F-impl.refine*[*OF PREFER-id-D*]

schematic-lemma *pn-map-impl-aux*:
shows (*?c*,*pn-map*) ∈
⟨⟨*Rm*,*Id*⟩*node-rel*⟩*list-set-rel*
→ ⟨⟨*nat-rel*,⟨*Id*⟩*list-set-rel* ×$_r$ ⟨*Id*⟩*list-set-rel*⟩*iam-map-rel*⟩*nres-rel*

unfolding *pn-map-def*[*abs-def*] *pn-props-def*
using [[*autoref-trace-failed-id*]]
apply (*autoref* (*keep-goal*, *trace*))
done

concrete-definition *pn-map-impl* **uses** *pn-map-impl-aux*
lemma *pn-map-impl-autoref*[*autoref-rules*]:
assumes *PREFER-id R*
shows (*pn-map-impl*,*pn-map*) ∈
⟨⟨*Rm*,*R*⟩*node-rel*⟩*list-set-rel*
→ ⟨⟨*nat-rel*,⟨*R*⟩*list-set-rel* ×$_r$ ⟨*R*⟩*list-set-rel*⟩*iam-map-rel*⟩*nres-rel*
using *assms pn-map-impl.refine* **by** *simp*

schematic-lemma *pn-map-code-aux*: *RETURN ?c* ≤ *pn-map-impl x*
unfolding *pn-map-impl-def*
apply (*refine-transfer* (*post*))
done
concrete-definition *pn-map-code* **uses** *pn-map-code-aux*
lemmas [*refine-transfer*] = *pn-map-code.refine*

thm *autoref-tyrel*

schematic-lemma *cr-rename-gba-impl-aux*:
assumes *ID*[*relator-props*]: *R*=*Id*
notes [*autoref-tyrel del*] = *TYRELI*[*of* ⟨*nat-rel*⟩*dflt-rs-rel*]
shows (*?c*,*cr-rename-gba*) ∈
⟨⟨*Rm*,*R*⟩*node-rel*⟩*list-set-rel* → ⟨*R*⟩*ltln-rel* → (*?R*::(*?'c* × -) *set*)
unfolding *ID*
unfolding *cr-rename-gba-def*[*abs-def*] *expand-init-def comp-def*
using [[*autoref-trace-failed-id*]]
apply (*autoref* (*keep-goal*, *trace*))
done
concrete-definition *cr-rename-gba-impl* **uses** *cr-rename-gba-impl-aux*

thm *cr-rename-gba-impl.refine*

lemma *cr-rename-gba-autoref*[*autoref-rules*]:
assumes *PREFER-id R*
shows (*cr-rename-gba-impl*, *cr-rename-gba*) ∈
⟨⟨*Rm*, *R*⟩*node-rel*⟩*list-set-rel* → ⟨*R*⟩*ltln-rel* →
⟨*gbav-impl-rel-ext unit-rel nat-rel* (⟨*R*⟩*fun-set-rel*)⟩*nres-rel*
using *assms cr-rename-gba-impl.refine*[*of R Rm*] **by** *simp*

schematic-lemma *cr-rename-gba-code-aux*: *RETURN ?c* ≤ *cr-rename-gba-impl*

x y
unfolding *cr-rename-gba-impl-def*
apply (*refine-transfer* (*post*))
done
concrete-definition *cr-rename-gba-code* **uses** *cr-rename-gba-code-aux*
lemmas [*refine-transfer*] = *cr-rename-gba-code.refine*

6.3.2 Creation of Graph

The implementation of the node-set. The relation enforces that there are no different nodes with the same name. This effectively establishes an additional invariant, made explicit by an assertion in the refined program. This invariant allows for a more efficient implementation.

definition *ls-nds-rel-def-internal*:
ls-nds-rel R ≡ ⟨*R*⟩*list-set-rel* ∩ {(-,*s*). *inj-on name s*}
lemma *ls-nds-rel-def*: ⟨*R*⟩*ls-nds-rel* = ⟨*R*⟩*list-set-rel* ∩ {(-,*s*). *inj-on name s*}
by (*simp add*: *relAPP-def ls-nds-rel-def-internal*)

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of ls-nds-rel i-set*]

lemma *ls-nds-rel-sv*[*relator-props*]:
assumes *single-valued R*
shows *single-valued* (⟨*R*⟩*ls-nds-rel*)
using *list-set-rel-sv*[*OF assms*]
unfolding *ls-nds-rel-def*
by (*metis inf.cobounded1 single-valued-subset*)

context begin interpretation *autoref-syn* **.**
lemma *lsnds-empty-autoref*[*autoref-rules*]:
assumes *PREFER-id R*
shows ([],{})∈⟨*R*⟩*ls-nds-rel*
using *assms*
apply (*simp add*: *ls-nds-rel-def*)
by *autoref*

lemma *lsnds-insert-autoref*[*autoref-rules*]:
assumes *SIDE-PRECOND* (*name n* ∉ *name*‘*ns*)
assumes (*n*′,*n*)∈*R*
assumes (*ns*′,*ns*)∈⟨*R*⟩*ls-nds-rel*
shows (*n*′#*ns*′,(*OP insert* ::: *R* → ⟨*R*⟩*ls-nds-rel* → ⟨*R*⟩*ls-nds-rel*)$*n*$*ns*)
∈ ⟨*R*⟩*ls-nds-rel*
using *assms*
unfolding *ls-nds-rel-def*
apply *simp*
proof (*elim conjE*, *rule conjI*)
assume [*autoref-rules*]: (*n*′, *n*) ∈ *R* (*ns*′, *ns*) ∈ ⟨*R*⟩*list-set-rel*
assume *name n* ∉ *name* ‘ *ns*
and *inj-on name ns*

hence $n \notin ns$ **by** (*auto*)
thus $(n' \# ns', insert\ n\ ns) \in \langle R \rangle list\text{-}set\text{-}rel$
by *autoref*
qed *auto*

lemma *ls-nds-image-autoref-aux*:
assumes [*autoref-rules*]: $(fi,f) \in Ra \to Rb$
assumes $(l,s) \in \langle Ra \rangle ls\text{-}nds\text{-}rel$
assumes [*simp*]: $\forall x.\ name\ (f\ x) = name\ x$
shows $(map\ fi\ l,\ f`s) \in \langle Rb \rangle ls\text{-}nds\text{-}rel$
proof −
from *assms* **have**
[*autoref-rules*]: $(l,s) \in \langle Ra \rangle list\text{-}set\text{-}rel$
and *INJ*: (*inj-on name s*)
by (*auto simp add*: *ls-nds-rel-def*)

have [*simp*]: *inj-on f s*
by (*rule inj-onI*) (*metis INJ assms*(*3*) *inj-on-eq-iff*)

have $(map\ fi\ l,\ f`s) \in \langle Rb \rangle list\text{-}set\text{-}rel$
by (*autoref* (*keep-goal*))
moreover have *inj-on name* (*f`s*)
apply (*rule inj-onI*)
apply (*auto simp*: *image-iff dest*: *inj-onD*[*OF INJ*])
done
ultimately show *?thesis*
by (*auto simp*: *ls-nds-rel-def*)
qed

lemma *ls-nds-image-autoref*[*autoref-rules*]:
assumes $(fi,f) \in Ra \to Rb$
assumes *SIDE-PRECOND* $(\forall x.\ name\ (f\ x) = name\ x)$
shows $(map\ fi,\ (OP\ op\ `::: (Ra \to Rb) \to \langle Ra \rangle ls\text{-}nds\text{-}rel \to \langle Rb \rangle ls\text{-}nds\text{-}rel)\$f)$
$\in \langle Ra \rangle ls\text{-}nds\text{-}rel \to \langle Rb \rangle ls\text{-}nds\text{-}rel$
using *assms*
unfolding *autoref-tag-defs*
using *ls-nds-image-autoref-aux*
by *blast*

lemma *list-set-autoref-to-list*[*autoref-ga-rules*]:
shows *is-set-to-sorted-list* $(\lambda\text{- -}.\ True)\ R$ *ls-nds-rel id*
unfolding *is-set-to-list-def is-set-to-sorted-list-def ls-nds-rel-def*
it-to-sorted-list-def list-set-rel-def br-def
by *auto*

end

context begin interpretation *autoref-syn* **.**
lemma [*autoref-itype*]:
upd-incoming
$::_i$ $\langle Im,\ I\rangle_i$*i-node* $\rightarrow_i$ $\langle\langle Im',\ I\rangle_i$*i-node*$\rangle_i$*i-set* $\rightarrow_i$ $\langle\langle Im',\ I\rangle_i$*i-node*$\rangle_i$*i-set*
by *simp*
end

term *upd-incoming*
schematic-lemma *upd-incoming-impl-aux*:
assumes *REL-IS-ID R*
shows (*?c*, *upd-incoming*)$\in\langle$*Rm1*,*R*$\rangle$*node-rel*
$\rightarrow \langle\langle$*Rm2*,*R*$\rangle$*node-rel*$\rangle$*ls-nds-rel*
$\rightarrow \langle\langle$*Rm2*,*R*$\rangle$*node-rel*$\rangle$*ls-nds-rel*
using *assms* **apply** *simp*
unfolding *upd-incoming-def*[*abs-def*]
using [[*autoref-trace-failed-id*]]
apply (*autoref* (*keep-goal*))
done

concrete-definition *upd-incoming-impl* **uses** *upd-incoming-impl-aux*
lemmas [*autoref-rules*] = *upd-incoming-impl.refine*[*OF PREFER-D*[*of REL-IS-ID*]]

schematic-lemma *expand-impl-aux*: (*?c*, *expand-aimpl*) $\in$
$\langle$*unit-rel*,*Id*$\rangle$*node-rel* $\times_r$ $\langle\langle$*unit-rel*,*Id*$\rangle$*node-rel*$\rangle$*ls-nds-rel*
$\rightarrow \langle$*nat-rel* $\times_r$ $\langle\langle$*unit-rel*,*Id*$\rangle$*node-rel*$\rangle$*ls-nds-rel*$\rangle$*nres-rel*
unfolding *expand-aimpl-def*[*abs-def*] *expand-new-name-def*
using [[*autoref-trace-failed-id*, *autoref-trace-intf-unif*]]
apply (*autoref* (*trace*,*keep-goal*))
done

concrete-definition *expand-impl* **uses** *expand-impl-aux*

context begin interpretation *autoref-syn* **.**
lemma [*autoref-itype*]: $expand_T$
$::_i$ $\langle\langle$*i-unit*, *I*$\rangle_i$*i-node*, $\langle\langle$*i-unit*, *I*$\rangle_i$*i-node*$\rangle_i$*i-set*$\rangle_i$*i-prod*
$\rightarrow_i$ $\langle\langle$*i-nat*,$\langle\langle$*i-unit*, *I*$\rangle_i$*i-node*$\rangle_i$*i-set*$\rangle_i$*i-prod*$\rangle_i$*i-nres* **by** *simp*

lemma *expand-autoref*[*autoref-rules*]:
assumes *ID*: *PREFER-id R*
assumes *A*: (*n-ns'*, *n-ns*)
$\in$ $\langle$*unit-rel*,*R*$\rangle$*node-rel* $\times_r$ $\langle\langle$*unit-rel*,*R*$\rangle$*node-rel*$\rangle$*list-set-rel*
assumes *B*: *SIDE-PRECOND* (
let (*n*,*ns*)=*n-ns in inj-on name ns* $\wedge$ ($\forall$ *n'*$\in$*ns*. *name n* $>$ *name n'*)
)
shows (*expand-impl n-ns'*,
(*OP expand-aimpl*
::: $\langle$*unit-rel*,*R*$\rangle$*node-rel* $\times_r$ $\langle\langle$*unit-rel*,*R*$\rangle$*node-rel*$\rangle$*list-set-rel*
$\rightarrow \langle$*nat-rel* $\times_r$ $\langle\langle$*unit-rel*,*R*$\rangle$*node-rel*$\rangle$*list-set-rel*$\rangle$*nres-rel*)$n-ns)

```
  ∈ ⟨nat-rel ×r ⟨⟨unit-rel,R⟩node-rel⟩list-set-rel⟩nres-rel
proof simp
  from ID A B have
    1: (n-ns, n-ns) ∈ Id ∩ {(-,(n,ns)). ∀ n'∈ns. name n > name n'}
    and 2: (n-ns', n-ns)
      ∈ ⟨unit-rel,Id⟩node-rel ×r ⟨⟨unit-rel,Id⟩node-rel⟩ls-nds-rel
    unfolding ls-nds-rel-def
    apply −
    apply auto []
    apply (cases n-ns')
    apply auto []
    done

  have [simp]: single-valued (nat-rel ×r ⟨⟨unit-rel, Id⟩node-rel⟩list-set-rel)
    by tagged-solver


  have [relator-props]: ⋀R. single-valued (Id ∩ R)
    by (metis IntE single-valuedD single-valuedI single-valued-Id)

  have [simp]: single-valued ((nat-rel ×r ⟨⟨unit-rel, Id⟩node-rel⟩ls-nds-rel) O
                        (Id ∩ {(-, n, ns). ∀ n'∈ns. name n' < n}))
    by (tagged-solver)

  from expand-impl.refine[THEN fun-relD, OF 2, THEN nres-relD]
  show (expand-impl n-ns', expand-aimpl n-ns)
    ∈ ⟨nat-rel ×r ⟨⟨unit-rel, R⟩node-rel⟩list-set-rel⟩nres-rel
    apply −
    apply (rule nres-relI)
    using ID
    apply (simp add: pw-le-iff refine-pw-simps)
    apply (fastforce simp: ls-nds-rel-def)
    done
qed

end


schematic-lemma expand-code-aux: RETURN ?c ≤ expand-impl n-ns
  unfolding expand-impl-def
  by (refine-transfer the-resI)
concrete-definition expand-code uses expand-code-aux
prepare-code-thms expand-code-def
lemmas [refine-transfer] = expand-code.refine


schematic-lemma create-graph-impl-aux:
  assumes ID: R=Id
  shows (?c, create-graph-aimpl)
```

$\in \langle R\rangle ltln\text{-}rel \to \langle\langle\langle unit\text{-}rel,R\rangle node\text{-}rel\rangle list\text{-}set\text{-}rel\rangle nres\text{-}rel$
unfolding *ID*
unfolding *create-graph-aimpl-def*[*abs-def*] *expand-init-def expand-new-name-def*
using [[*autoref-trace-failed-id*]]
apply (*autoref* (*keep-goal*))
done
concrete-definition *create-graph-impl* **uses** *create-graph-impl-aux*

lemmas [*autoref-rules*] = *create-graph-impl.refine*[*OF PREFER-id-D*]

schematic-lemma *create-graph-code-aux*: *RETURN ?c* ≤ *create-graph-impl* φ
unfolding *create-graph-impl-def*
by *refine-transfer*
concrete-definition *create-graph-code* **uses** *create-graph-code-aux*
lemmas [*refine-transfer*] = *create-graph-code.refine*

schematic-lemma *create-name-gba-impl-aux*:
(*?c*, (*create-name-gba-aimpl*:: *'a*::*linorder ltln* ⇒ -))
$\in \langle Id\rangle ltln\text{-}rel \to$ (*?R*::(*?'c*×-) *set*)
unfolding *create-name-gba-aimpl-def*[*abs-def*]
using [[*autoref-trace-failed-id*]]
apply (*autoref* (*keep-goal*, *trace*))
done
concrete-definition *create-name-gba-impl* **uses** *create-name-gba-impl-aux*

lemma *create-name-gba-autoref*[*autoref-rules*]:
assumes *PREFER-id R*
shows
(*create-name-gba-impl*, *create-name-gba*)
$\in \langle R\rangle ltln\text{-}rel \to \langle gbav\text{-}impl\text{-}rel\text{-}ext\ unit\text{-}rel\ nat\text{-}rel\ (\langle R\rangle fun\text{-}set\text{-}rel)\rangle nres\text{-}rel$
(**is** -:-→⟨*?R*⟩*nres-rel*)
proof (*intro fun-relI nres-relI*)
fix φ φ'
assume *A*: (φ,φ')∈⟨*R*⟩*ltln-rel*
from *assms* **have** *RID*[*simp*]: *R*=*Id* **by** *simp*

note *create-name-gba-impl.refine*[*THEN fun-relD*,*THEN nres-relD*, *OF A*[*unfolded RID*]]
also note *create-name-gba-aimpl-refine*
finally show *create-name-gba-impl* φ ≤ ⇓ *?R* (*create-name-gba* φ') **by** *simp*
qed

schematic-lemma *create-name-gba-code-aux*: *RETURN ?c* ≤ *create-name-gba-impl* φ
unfolding *create-name-gba-impl-def*
by (*refine-transfer* (*post*))
concrete-definition *create-name-gba-code* **uses** *create-name-gba-code-aux*
lemmas [*refine-transfer*] = *create-name-gba-code.refine*

declare [[*show-types*, *show-sorts*, *show-consts*]]
ML-val ‹@{*thm autoref-itype*(*1*)} |> *Thm.prop-of*›
term *create-name-igba*

schematic-lemma *create-name-igba-impl-aux*:
  assumes *RID*: *R*=*Id*
  shows (*?c*,*create-name-igba*)∈
  ⟨*R*⟩*ltln-rel* → ⟨*igbav-impl-rel-ext unit-rel nat-rel* (⟨*R*⟩*fun-set-rel*)⟩*nres-rel*
  unfolding *RID*
  unfolding *create-name-igba-def*[*abs-def*]
  using [[*autoref-trace-failed-id*]]
  apply (*autoref* (*trace*, *keep-goal*))
  done
concrete-definition *create-name-igba-impl* **uses** *create-name-igba-impl-aux*
lemmas [*autoref-rules*] = *create-name-igba-impl.refine*[*OF PREFER-id-D*]

schematic-lemma *create-name-igba-code-aux*: *RETURN ?c* ≤ *create-name-igba-impl* φ
  unfolding *create-name-igba-impl-def*
  by (*refine-transfer* (*post*))
concrete-definition *create-name-igba-code* **uses** *create-name-igba-code-aux*
lemmas [*refine-transfer*] = *create-name-igba-code.refine*

export-code *create-name-igba-code* **checking** *SML*

end

References

[1] J. Esparza, P. Lammich, R. Neumann, T. Nipkow, A. Schimpf, and J.-G. Smaus. A fully verified executable ltl model checker. In N. Sharygina and H. Veith, editors, *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 463–478. Springer Berlin Heidelberg, 2013.

[2] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 3–18, London, UK, UK, 1996. Chapman & Hall, Ltd.

[3] S. Merz. Stuttering equivalence. *Archive of Formal Proofs*, May 2012. http://afp.sf.net/entries/Stuttering_Equivalence.shtml, Formal proof development.