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 **3**

2 Linear Temporal Logic **3**
2.1 LTL formulas . . . 3
2.1.1 Syntax . . . 3
2.1.2 Semantics . . . 4
2.1.3 Explicit Syntactic Sugar . . . 4
2.2 Semantic Preserving Syntax Transformations . . . 7
2.3 LTL formula in negation normal form (NNF) . . . 10

3 Rewriting LTL formulas **13**

4 Stutter Invariance of next-free LTL Formula **17**

5 LTL to GBA translation **19**
5.1 Statistics . . . 19
5.2 Preliminaries . . . 20
5.3 Creation of States . . . 21
5.4 Creation of GBA . . . 30

6 Refinement to Efficient Code **39**
6.1 Parametricity Setup Boilerplate . . . 39
6.1.1 LTL Formulas . . . 39
6.1.2 Nodes . . . 42
6.2 Massaging the Abstract Algorithm . . . 45
6.2.1 Creation of the Nodes . . . 45
6.2.2 Creation of GBA from Nodes . . . 47
6.3 Refinement to Efficient Data Structures . . . 50
6.3.1 Creation of GBA from Nodes . . . 50
6.3.2 Creation of Graph . . . 52

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* ⟨*proof*⟩

primrec *ltl-semantics* :: *'ap set word* ⇒ *'ap ltl* ⇒ *bool*
(- ⊨ - [*80*,*80*] *80*)
where
$\xi \models \mathit{true} = \mathit{True}$
| $\xi \models \mathit{false} = \mathit{False}$
| $\xi \models \mathit{prop}(q) = (q \in (\xi\ 0))$
| $\xi \models \mathit{not}\ \varphi = (\neg\ \xi \models \varphi)$
| $\xi \models \varphi\ \mathit{and}\ \psi = (\xi \models \varphi \wedge \xi \models \psi)$
| $\xi \models \varphi\ \mathit{or}\ \psi = (\xi \models \varphi \vee \xi \models \psi)$
| $\xi \models X\ \varphi = (\mathit{suffix}\ 1\ \xi \models \varphi)$
| $\xi \models \varphi\ U\ \psi = (\exists i.\ \mathit{suffix}\ i\ \xi \models \psi \wedge (\forall j{<}i.\ \mathit{suffix}\ j\ \xi \models \varphi))$
| $\xi \models \varphi\ V\ \psi = (\forall i.\ \mathit{suffix}\ i\ \xi \models \psi \vee (\exists j{<}i.\ \mathit{suffix}\ j\ \xi \models \varphi))$

definition *ltl-language* $\varphi \equiv \{\xi.\ \xi \models \varphi\}$
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* ⟨*proof*⟩

primrec *ltlc-semantics*
:: [*'a set word*, *'a ltlc*] ⇒ *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
⟨*proof*⟩

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

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″*
⟨*proof*⟩

lemma *ltlc-eq-on*: *pw-eq-on* (*ltlc-aprops* φ) *w w′* ⟹ *w* ⊨*c* φ ⟷ *w′* ⊨*c* φ
⟨*proof*⟩

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
⟨*proof*⟩

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*⟩

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 ⟷ (λ*i*. (*the o f*) −‘ ξ *i*) ⊨*c* φ
⟨*proof*⟩

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* $(\varphi\ U_c\ \psi)$ = *ltlc-to-ltl* φ *U ltlc-to-ltl* ψ
| *ltlc-to-ltl* $(\varphi\ V_c\ \psi)$ = *ltlc-to-ltl* φ *V ltlc-to-ltl* ψ

lemma *ltlc-to-ltl-equiv*:
$\xi \models$ (*ltlc-to-ltl* φ) $\longleftrightarrow\ \xi \models_c \varphi$
⟨*proof*⟩

end

2.2 Semantic Preserving Syntax Transformations

context begin interpretation *LTL-Syntax* ⟨*proof*⟩

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

lemma *ltl-false-true-con*[*simp*]:
$\xi \models$ *not true* $\longleftrightarrow\ \xi \models$ *false*
⟨*proof*⟩

The negation symbol can be passed through the next operator.

lemma *ltl-Next-Neg-con*[*simp*]:
$\xi \models X$ (*not* φ) $\longleftrightarrow\ \xi \models$ *not* $X\ \varphi$
⟨*proof*⟩

The connection between Until and Release

lemma *ltl-Release-Until-con*:
$\xi \models \varphi\ V\ \psi \longleftrightarrow (\neg\ \xi \models$ (*not* φ) U (*not* ψ))
⟨*proof*⟩

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*⟩

lemma *ltl-expand-Release*:
$\xi \models \varphi\ V\ \psi \longleftrightarrow (\xi \models \psi$ *and* $(\varphi$ *or* $(X\ (\varphi\ V\ \psi))))$
⟨*proof*⟩

Double negation structure of an LTL formula

lemma [*simp*]:
not $((\lambda\mu.$ *not not* $\mu)\ \hat{}\hat{}\ n)\ \varphi = ((\lambda\mu.$ *not not* $\mu)\ \hat{}\hat{}\ n)$ (*not* φ)
⟨*proof*⟩

lemma *ltl-double-neg-struct*:
shows $\exists n\ \psi.\ \varphi = ((\lambda\xi.$ *not not* $\xi)\ \hat{}\hat{}\ n)\ \psi \wedge (\forall \nu.\ \psi \neq$ *not not* $\nu)$
(**is** $\exists n\ \psi.$ *?Q* $\varphi\ n\ \psi$)
⟨*proof*⟩

lemma *ltl-size-double-neg*:
assumes $\psi = ((\lambda\mu.\ not\ not\ \mu)\ \hat{}\hat{}\ n)\ \varphi$
shows $size\ \varphi \leq size\ \psi$
⟨*proof*⟩

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 $\xi \models$ *ltl-pushneg* (*not* φ) $\longleftrightarrow \xi \models$ *not ltl-pushneg* φ
⟨*proof*⟩

theorem *ltl-pushneg-equiv*[*simp*]:
$\xi \models$ *ltl-pushneg* $\varphi \longleftrightarrow \xi \models \varphi$
⟨*proof*⟩

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
⟨*proof*⟩

lemma *ltl-pushneg-neg-struct*:
assumes *ltl-pushneg* φ = *not* ψ
shows ∃ *q*. ψ = *prop*(*q*)
⟨*proof*⟩

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
⟨*proof*⟩

lemma *subformula-size*:
  assumes ψ *is-subformula-of* φ
    shows *size* $\psi <$ *size* $\varphi \vee \psi = \varphi$
⟨*proof*⟩

lemma *subformula-on-ltl-pushneg*:
  assumes ψ *is-subformula-of* (*ltl-pushneg* φ)
    shows $\exists \mu.$ $\psi =$ *ltl-pushneg* μ
⟨*proof*⟩

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 $\exists q.$ $\psi =$ *prop*(q)
⟨*proof*⟩

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
⟨*proof*⟩

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*⟩

lemma *ltl-pushneg-size-lin-help*:
  assumes ψ = *ltl-pushneg* φ
    shows *size* ψ + *1* ≤ *size* φ + *leafcnt* φ
⟨*proof*⟩

theorem *ltl-pushneg-size-lin*:
  size (*ltl-pushneg* φ) ≤ *2* ∗ *size* φ
⟨*proof*⟩

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* ⟨*proof*⟩

primrec *ltln-semantics* :: [*'a set word*, *'a ltln*] ⇒ *bool*
(- $\models_n$ - [80,80] 80)
where
$\xi \models_n true_n = True$
| $\xi \models_n false_n = False$
| $\xi \models_n prop_n(q) = (q \in \xi\ 0)$
| $\xi \models_n nprop_n(q) = (q \notin \xi\ 0)$
| $\xi \models_n \varphi\ and_n\ \psi = (\xi \models_n \varphi \wedge \xi \models_n \psi)$
| $\xi \models_n \varphi\ or_n\ \psi = (\xi \models_n \varphi \vee \xi \models_n \psi)$
| $\xi \models_n X_n\ \varphi = (suffix\ 1\ \xi \models_n \varphi)$
| $\xi \models_n \varphi\ U_n\ \psi = (\exists i.\ suffix\ i\ \xi \models_n \psi \wedge (\forall j{<}i.\ suffix\ j\ \xi \models_n \varphi))$
| $\xi \models_n \varphi\ V_n\ \psi = (\forall i.\ suffix\ i\ \xi \models_n \psi \vee (\exists j{<}i.\ suffix\ j\ \xi \models_n \varphi))$

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* $(\varphi\ U\ \psi)$ = *ltl-to-ltln* $\varphi\ U_n$ *ltl-to-ltln* ψ
| *ltl-to-ltln* $(\varphi\ V\ \psi)$ = *ltl-to-ltln* $\varphi\ V_n$ *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
⟨*proof*⟩

lemma *ltl-nnf-equiv*[*simp*]:
$\xi \models_n$ *ltl-to-ltln* (*ltl-pushneg* ψ) $\longleftrightarrow \xi \models \psi$
⟨*proof*⟩

fun *subfrmlsn* :: *'a ltln* ⇒ *'a ltln set*
where
subfrmlsn $(\mu\ and_n\ \psi)$ = $\{\mu\ and_n\ \psi\} \cup$ *subfrmlsn* $\mu \cup$ *subfrmlsn* ψ
| *subfrmlsn* $(X_n\ \mu)$ = $\{X_n\ \mu\} \cup$ *subfrmlsn* μ
| *subfrmlsn* $(\mu\ U_n\ \psi)$ = $\{\mu\ U_n\ \psi\} \cup$ *subfrmlsn* $\mu \cup$ *subfrmlsn* ψ
| *subfrmlsn* $(\mu\ V_n\ \psi)$ = $\{\mu\ V_n\ \psi\} \cup$ *subfrmlsn* $\mu \cup$ *subfrmlsn* ψ
| *subfrmlsn* $(\mu\ or_n\ \psi)$ = $\{\mu\ or_n\ \psi\} \cup$ *subfrmlsn* $\mu \cup$ *subfrmlsn* ψ
| *subfrmlsn* x = $\{x\}$

lemma *subfrmlsn-id*[*simp*]: $\varphi \in$ *subfrmlsn* φ ⟨*proof*⟩
lemma *subfrmlsn-finite*: *finite* (*subfrmlsn* φ) ⟨*proof*⟩
lemma *subfrmlsn-subset*:$\psi \in$ *subfrmlsn* $\varphi \Longrightarrow$ *subfrmlsn* $\psi \subseteq$ *subfrmlsn* φ
⟨*proof*⟩

fun *size-frmln* :: *'a ltln* ⇒ *nat*
where
size-frmln $(\varphi\ and_n\ \psi)$ = *size-frmln* φ + *size-frmln* ψ + *1*
| *size-frmln* $(X_n\ \varphi)$ = *size-frmln* φ + *1*
| *size-frmln* $(\varphi\ U_n\ \psi)$ = *size-frmln* φ + *size-frmln* ψ + *1*
| *size-frmln* $(\varphi\ V_n\ \psi)$ = *size-frmln* φ + *size-frmln* ψ + *1*
| *size-frmln* $(\varphi\ or_n\ \psi)$ = *size-frmln* φ + *size-frmln* ψ + *1*
| *size-frmln* - = *1*

lemma *size-frmln-gt-zero*[*simp*]: *size-frmln* $\varphi > 0$ ⟨*proof*⟩

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*⟩

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*⟩

lemma *ltln-expand-Release*:
$\xi \models_n \varphi\ V_n\ \psi = (\xi \models_n \psi\ and_n\ (\varphi\ or_n\ (X_n\ (\varphi\ V_n\ \psi))))$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *ltln-Release-alterdef*:
$\xi \models_n \varphi\ V_n\ \psi \longleftrightarrow \xi \models_n (\Box_n\ \psi)\ or_n\ (\psi\ U_n\ (\varphi\ and_n\ \psi))$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

end
end

3 Rewriting LTL formulas

theory *LTL-Rewrite*
imports
LTL
begin

context begin interpretation *LTL-Syntax* ⟨*proof*⟩

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 $\xi \models_n \varphi\ U_n\ \psi \longleftrightarrow \xi \models_n \psi$
⟨*proof*⟩

corollary *ltln-pure-eventual-frmls-equiv-diamond*:
assumes $\psi \in$ *ltln-pure-eventual-frmls*
shows $\xi \models_n \Diamond_n\ \psi \longleftrightarrow \xi \models_n \psi$

⟨*proof*⟩

inductive-set *ltln-pure-universal-frmls* :: *'a ltln set*
where
  $\Box_n\ \varphi \in$ *ltln-pure-universal-frmls*
| ⟦ $\nu \in$ *ltln-pure-universal-frmls*; $\mu \in$ *ltln-pure-universal-frmls* ⟧
    ⟹ ν *and*$_n$ $\mu \in$ *ltln-pure-universal-frmls*
| ⟦ $\nu \in$ *ltln-pure-universal-frmls*; $\mu \in$ *ltln-pure-universal-frmls* ⟧
    ⟹ ν *or*$_n$ $\mu \in$ *ltln-pure-universal-frmls*
| ⟦ $\nu \in$ *ltln-pure-universal-frmls*; $\mu \in$ *ltln-pure-universal-frmls* ⟧
    ⟹ $\nu\ U_n\ \mu \in$ *ltln-pure-universal-frmls*
| ⟦ $\nu \in$ *ltln-pure-universal-frmls*; $\mu \in$ *ltln-pure-universal-frmls* ⟧
    ⟹ $\nu\ V_n\ \mu \in$ *ltln-pure-universal-frmls*

theorem *ltln-pure-universal-frmls-equiv*:
  assumes $\psi \in$ *ltln-pure-universal-frmls*
  shows $\xi \models_n \varphi\ V_n\ \psi \longleftrightarrow \xi \models_n \psi$
⟨*proof*⟩

Some simple rewrite rules

fun *ltln-rewrite-step* :: *'a ltln* ⇒ *'a ltln*
where
  ltln-rewrite-step (- U_n *true*$_n$) = *true*$_n$
| *ltln-rewrite-step* (- V_n *false*$_n$) = *false*$_n$
| *ltln-rewrite-step* (*true*$_n$ U_n (- U_n μ)) = *true*$_n$ U_n μ
| *ltln-rewrite-step* (*false*$_n$ V_n (- V_n μ)) = *false*$_n$ V_n μ
| *ltln-rewrite-step* ψ = (*case* ψ *of*
    $\varphi\ U_n\ \varphi'$ ⇒
      if $\varphi = \varphi'$ *then* φ
      else if $\varphi' \in$ *ltln-pure-eventual-frmls then* φ'
      else ψ
  | $\varphi\ V_n\ \varphi'$ ⇒
      if $\varphi = \varphi'$ *then* φ
      else if $\varphi' \in$ *ltln-pure-universal-frmls then* φ'
      else ψ
  | ($\varphi\ U_n\ \mu$) *and*$_n$ ($\nu\ U_n\ \mu'$) ⇒ *if* $\mu = \mu'$ *then* (φ *and*$_n$ ν) $U_n\ \mu$ *else* ψ
  | ($\varphi\ U_n\ \nu$) *or*$_n$ ($\varphi'\ U_n\ \mu$) ⇒ *if* $\varphi = \varphi'$ *then* $\varphi\ U_n$ (ν *or*$_n$ μ) *else* ψ
  | ($\varphi\ V_n\ \nu$) *and*$_n$ ($\varphi'\ V_n\ \mu$) ⇒ *if* $\varphi = \varphi'$ *then* $\varphi\ V_n$ (ν *and*$_n$ μ) *else* ψ
  | ($\varphi\ V_n\ \mu$) *or*$_n$ ($\nu\ V_n\ \mu'$) ⇒ *if* $\mu = \mu'$ *then* (φ *or*$_n$ ν) $V_n\ \mu$ *else* ψ
  | - ⇒ ψ)

lemma *ltln-rewrite-step--size-less*:
  assumes *ltln-rewrite-step* $\psi \neq \psi$
  shows *size* (*ltln-rewrite-step* ψ) < *size* ψ
  ⟨*proof*⟩

lemma *ltln-rewrite-step--size-leq*:
  size (*ltln-rewrite-step* ψ) ≤ *size* ψ

⟨*proof*⟩

theorem *ltln-rewrite-step--equiv*:
$\xi \models_n$ *ltln-rewrite-step* $\psi \longleftrightarrow \xi \models_n \psi$
⟨*proof*⟩

function *ltln-rewrite-rec*
where
ltln-rewrite-rec ψ = (*case ltln-rewrite-step* ψ *of*
ν *and*$_n$ $\mu \Rightarrow$ (*ltln-rewrite-rec* ν) *and*$_n$ (*ltln-rewrite-rec* μ)
| ν *or*$_n$ $\mu \Rightarrow$ (*ltln-rewrite-rec* ν) *or*$_n$ (*ltln-rewrite-rec* μ)
| X_n $\nu \Rightarrow X_n$ (*ltln-rewrite-rec* ν)
| ν U_n $\mu \Rightarrow$ (*ltln-rewrite-rec* ν) U_n (*ltln-rewrite-rec* μ)
| ν V_n $\mu \Rightarrow$ (*ltln-rewrite-rec* ν) V_n (*ltln-rewrite-rec* μ)
| $\varphi \Rightarrow \varphi$)
⟨*proof*⟩
termination ⟨*proof*⟩
thm *ltln-rewrite-step--size-leq*
⟨*proof*⟩

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

lemma *ltln-rewrite-rec--size-less*:
assumes *ltln-rewrite-rec* $\psi \neq \psi$
shows *size* (*ltln-rewrite-rec* ψ) < *size* ψ
⟨*proof*⟩

lemma *ltln-rewrite-rec--size-leq*:
size (*ltln-rewrite-rec* ψ) $\leq$ *size* ψ
⟨*proof*⟩

theorem *ltln-rewrite-rec--equiv*:
$\xi \models_n$ *ltln-rewrite-rec* $\psi \longleftrightarrow \xi \models_n \psi$
⟨*proof*⟩

function *ltln-rewrite*
where
ltln-rewrite ψ
= (*let* φ = *ltln-rewrite-rec* ψ *in*
if $\varphi \neq \psi$ *then ltln-rewrite* φ *else* ψ)
⟨*proof*⟩
termination
⟨*proof*⟩

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

lemma *ltln-rewrite--size-less*:
  assumes *ltln-rewrite* $\psi \neq \psi$
  shows *size* (*ltln-rewrite* ψ) < *size* ψ
⟨*proof*⟩

lemma *ltln-rewrite--size-leq*:
  size (*ltln-rewrite* ψ) ≤ *size* ψ
⟨*proof*⟩

lemma *ltln-rewrite--equiv*:
  $\xi \models_n$ *ltln-rewrite* $\psi \longleftrightarrow \xi \models_n \psi$
⟨*proof*⟩

fun *ltln-pure-eventual-frmls-impl*
where
  ltln-pure-eventual-frmls-impl ($\Diamond_n$ φ) = *True*
| *ltln-pure-eventual-frmls-impl* (ν *and*$_n$ 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*]:
  $\varphi \in$ *ltln-pure-eventual-frmls* $\longleftrightarrow$ *ltln-pure-eventual-frmls-impl* φ
  (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

fun *ltln-pure-universal-frmls-impl*
where
  ltln-pure-universal-frmls-impl ($\Box_n$ 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*]:
  $\varphi \in$ *ltln-pure-universal-frmls* $\longleftrightarrow$ *ltln-pure-universal-frmls-impl* φ

(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

definition
ltln-rewrite-step-impl $\psi \equiv$ *case* ψ *of*
$\nu\ U_n\ \mu \Rightarrow$ *if* $\mu = true_n$ *then* $true_n$
else (
case (ν, μ) *of*
($true_n$, - $U_n\ \mu'$) $\Rightarrow$ $true_n\ U_n\ \mu'$
| - $\Rightarrow$ *if* $\nu = \mu$ *then* ν
else if $\mu \in$ *ltln-pure-eventual-frmls then* μ
else ψ)
| $\nu\ V_n\ \mu \Rightarrow$ *if* $\mu = false_n$ *then* $false_n$
else (
case (ν, μ) *of*
($false_n$, - $V_n\ \mu'$) $\Rightarrow$ $false_n\ V_n\ \mu'$
| - $\Rightarrow$ *if* $\nu = \mu$ *then* ν
else if $\mu \in$ *ltln-pure-universal-frmls then* μ
else ψ)
| $\psi 1\ and_n\ \psi 2 \Rightarrow$ (
case ($\psi 1$, $\psi 2$) *of*
($\varphi\ U_n\ \mu$, $\nu\ U_n\ \mu'$) $\Rightarrow$ *if* $\mu = \mu'$ *then* $(\varphi\ and_n\ \nu)\ U_n\ \mu$ *else* ψ
| ($\varphi\ V_n\ \nu$, $\varphi'\ V_n\ \mu$) $\Rightarrow$ *if* $\varphi = \varphi'$ *then* $\varphi\ V_n\ (\nu\ and_n\ \mu)$ *else* ψ
| - $\Rightarrow$ ψ)
| $\psi 1\ or_n\ \psi 2 \Rightarrow$ (
case ($\psi 1$, $\psi 2$) *of*
($\varphi\ U_n\ \nu$, $\varphi'\ U_n\ \mu$) $\Rightarrow$ *if* $\varphi = \varphi'$ *then* $\varphi\ U_n\ (\nu\ or_n\ \mu)$ *else* ψ
| ($\varphi\ V_n\ \mu$, $\nu\ V_n\ \mu'$) $\Rightarrow$ *if* $\mu = \mu'$ *then* $(\varphi\ or_n\ \nu)\ V_n\ \mu$ *else* ψ
| - $\Rightarrow$ ψ)
| - $\Rightarrow$ ψ

lemma *ltln-rewrite-step-unfold*[*code-unfold*]:
ltln-rewrite-step = *ltln-rewrite-step-impl*
⟨*proof*⟩

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*)
⟨*proof*⟩

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) ⟷ *ltl-semantics r* φ
⟨*proof*⟩

lemma *PLTL-next-free-cnv*[*simp*]: *PLTL.next-free* (*cnv* φ) ⟷ *ltl-next-free* φ
⟨*proof*⟩

theorem *next-free-stutter-invariant*:
assumes *next-free*: *ltl-next-free* φ
assumes *eq*: *r* ≈ *r'*

shows $r \models \varphi \longleftrightarrow r' \models \varphi$
— A next free formula cannot distinguish between stutter-equivalent runs.
⟨*proof*⟩

context begin interpretation *LTL-Syntax* ⟨*proof*⟩
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* $(not_c\ \varphi)$ = *ltlc-next-free* φ
| *ltlc-next-free* $(\varphi\ and_c\ \psi)$ = (*ltlc-next-free* $\varphi \wedge$ *ltlc-next-free* ψ)
| *ltlc-next-free* $(\varphi\ or_c\ \psi)$ = (*ltlc-next-free* $\varphi \wedge$ *ltlc-next-free* ψ)
| *ltlc-next-free* $(\varphi\ implies_c\ \psi)$ = (*ltlc-next-free* $\varphi \wedge$ *ltlc-next-free* ψ)
| *ltlc-next-free* $(\varphi\ iff_c\ \psi)$ = (*ltlc-next-free* $\varphi \wedge$ *ltlc-next-free* ψ)
| *ltlc-next-free* $(X_c\ \varphi)$ = *False*
| *ltlc-next-free* $(F_c\ \varphi)$ = *ltlc-next-free* φ
| *ltlc-next-free* $(G_c\ \varphi)$ = *ltlc-next-free* φ
| *ltlc-next-free* $(\varphi\ U_c\ \psi)$ = (*ltlc-next-free* $\varphi \wedge$ *ltlc-next-free* ψ)
| *ltlc-next-free* $(\varphi\ V_c\ \psi)$ = (*ltlc-next-free* $\varphi \wedge$ *ltlc-next-free* ψ)
end

lemma *ltlc-to-ltl-next-free-iff*:
ltl-next-free (*ltlc-to-ltl* φ) $\longleftrightarrow$ *ltlc-next-free* φ
⟨*proof*⟩

theorem *ltlc-next-free-stutter-invariant*:
assumes *next-free*: *ltlc-next-free* φ
assumes *eq*: $r \approx r'$
shows $r \models_c \varphi \longleftrightarrow r' \models_c \varphi$
— A next free formula cannot distinguish between stutter-equivalent runs.
⟨*proof*⟩

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
  ⟨proof⟩

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 ⟨proof⟩
```

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)
  ⟨proof⟩
```

lemma *SPEC-rule-param2*:
assumes $f\ x \leq SPEC\ (P\ x)$
and $\bigwedge r1\ r2.\ (P\ x)\ (r1,\ r2) \Longrightarrow (P\ x')\ (r1,\ r2)$
shows $f\ x \leq SPEC\ (P\ x')$
⟨*proof*⟩

lemma *SPEC-rule-weak*:
assumes $f\ x \leq SPEC\ (Q\ x)$ **and** $f\ x \leq SPEC\ (P\ x)$
and $\bigwedge r1\ r2.\ [\![(Q\ x)\ (r1,\ r2);\ (P\ x)\ (r1,\ r2)]\!] \Longrightarrow (P\ x')\ (r1,\ r2)$
shows $f\ x \leq SPEC\ (P\ x')$
⟨*proof*⟩

lemma *SPEC-rule-weak-nested2*: $[\![f \leq SPEC\ Q;\ f \leq SPEC\ P;$
$\bigwedge r1\ r2.\ [\![Q\ (r1,\ r2);\ P\ (r1,\ r2)]\!] \Longrightarrow g\ (r1,\ r2) \leq SPEC\ P]\!]$
$\Longrightarrow f \leq SPEC\ (\lambda r'.\ g\ r' \leq SPEC\ P)$
⟨*proof*⟩

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* ⟨*proof*⟩

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* $\equiv$ *0*
definition *expand-new-name* $\equiv$ *Suc*

lemma *expand-new-name-expand-init*:
  shows $\forall$ *nm*. *expand-init* $<$ *expand-new-name nm*
⟨*proof*⟩

lemma *expand-new-name-step*[*intro*]:
  shows $\bigwedge$*n*. *name* (*n*::'*a node*) $<$ *expand-new-name* (*name n*)
⟨*proof*⟩

lemma *expand-new-name--less-zero*[*intro*]: *0* $<$ *expand-new-name nm*
⟨*proof*⟩

abbreviation
  upd-incoming-f n $\equiv$ (λ*n'*.
    if (*old n'* = *old n* $\wedge$ *next n'* = *next n*) *then*
      n'(| *incoming* := *incoming n* $\cup$ *incoming n'* |)
    else n'
)

definition
 upd-incoming n ns $\equiv$ ((*upd-incoming-f n*) ‘ *ns*)

lemma *upd-incoming--elem*:
  assumes *nd*$\in$*upd-incoming n ns*
    shows *nd*$\in$*ns*
        $\vee$ ($\exists$ *nd'*$\in$*ns*. *nd* = *nd'*(| *incoming* := *incoming n* $\cup$ *incoming nd'* |) $\wedge$
                        old nd' = *old n* $\wedge$
                        next nd' = *next n*)

⟨*proof*⟩

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*⟩

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*⟩

lemma *name-upd-incoming-f*[*simp*]: *name* (*upd-incoming-f n x*) = *name x*
⟨*proof*⟩

lemma *name-upd-incoming*[*simp*]:
name ‘ (*upd-incoming n ns*) = *name* ‘ *ns* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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* ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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
$\equiv (\forall\, q{\in}snd\ nm\text{-}nds.\ (fst\ nm\text{-}nds > name\ q \wedge (\forall\, nm'{\in}incoming\ q.\ fst\ nm\text{-}nds > nm')))$

abbreviation
expand-rslt-name n-ns nm-nds
$\equiv (name\ (fst\ n\text{-}ns) \leq fst\ nm\text{-}nds \wedge name\ `\ (snd\ n\text{-}ns) \subseteq name\ `\ (snd\ nm\text{-}nds))$
$\wedge\ name\ `\ (snd\ nm\text{-}nds)$
$= name\ `\ (snd\ n\text{-}ns) \cup name\ `\ \{nd{\in}snd\ nm\text{-}nds.\ name\ nd \geq name\ (fst\ n\text{-}ns)\}$

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

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

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

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*))
⟨*proof*⟩

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* $\leq$ *SPEC* (*expand-rslt-exist-eq n-ns*)
(**is** - $\leq$ *SPEC* (*?P n-ns*))
⟨*proof*⟩

lemma *expand-prop-exist*:
shows *expand n-ns*
$\leq$ *SPEC* (λr. *expand-assm-exist* ξ *n-ns* $\longrightarrow$ *expand-rslt-exist* ξ *n-ns r*)
(**is** - $\leq$ *SPEC* (*?P n-ns*))
⟨*proof*⟩

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)
⟨*proof*⟩

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)
⟨*proof*⟩

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* $\{\varphi\}$
⟨*proof*⟩

lemma *new2-less-setsum*: *frmlset-sumn* (*new2* φ) $<$ *frmlset-sumn* $\{\varphi\}$
⟨*proof*⟩

lemma *new1-finite*[*intro*]:*finite* (*new1* ψ) ⟨*proof*⟩
lemma *new1-subset-frmls*:$\varphi \in$ *new1* $\psi \Longrightarrow \varphi \in$ *subfrmlsn* ψ ⟨*proof*⟩

lemma *new2-finite*[*intro*]:*finite* (*new2* ψ) ⟨*proof*⟩
lemma *new2-subset-frmls*:$\varphi \in$ *new2* $\psi \Longrightarrow \varphi \in$ *subfrmlsn* ψ ⟨*proof*⟩

lemma *next1-finite*[*intro*]:*finite* (*next1* ψ) ⟨*proof*⟩
lemma *next1-subset-frmls*:$\varphi \in$ *next1* $\psi \Longrightarrow \varphi \in$ *subfrmlsn* ψ ⟨*proof*⟩

lemma *expand-inv-result* φ *ns* $\Longrightarrow$ *old-next-pair* ‘ *ns* $\subseteq$ *old-next-limit* φ
⟨*proof*⟩

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* − $\{\psi\}$, *old* := $\{\psi\}$ $\cup$ *old n* ⦈) $\vee$
(n' = n⦇ *new* := *new1* ψ $\cup$ (*new n* − $\{\psi\}$),
old := $\{\psi\}$ $\cup$ *old n*,
next := *next1* ψ $\cup$ *next n* ⦈) $\vee$
(n' = n⦇ *name* := *nm*,
new := *new2* ψ $\cup$ (*new n* − $\{\psi\}$), *old* := $\{\psi\}$ $\cup$ *old n* ⦈)
(**is** *?case1* $\vee$ *?case2* $\vee$ *?case3*)
shows ((n', *ns*′), (n, *ns*))$\in$*expand-ord* φ $\wedge$ *expand-inv* φ (n', *ns*′)
(**is** *?concl1* $\wedge$ *?concl2*)
⟨*proof*⟩

lemma *expand-term-prop-help*:
assumes ((n', *ns*′), (n, *ns*))$\in$*expand-ord* φ $\wedge$ *expand-inv* φ (n', *ns*′)
and *assm-rule*: ⟦*expand-inv* φ (n', *ns*′); ((n', *ns*′), n, *ns*) $\in$ *expand-ord* φ⟧
$\Longrightarrow$ f (n', *ns*′) $\leq$ *SPEC P*
shows f (n', *ns*′) $\leq$ *SPEC P*
⟨*proof*⟩

lemma *expand-inv-upd-incoming*:

assumes *expand-inv* φ (*n*, *ns*)
shows *expand-inv-result* φ (*upd-incoming n ns*)
⟨*proof*⟩

lemma *upd-incoming-eq-old-next-pair*:
shows *old-next-pair* ‘ *ns* = *old-next-pair* ‘ (*upd-incoming n ns*) (**is** *?A* = *?B*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *expand-eq-expand*$_T$:
assumes *I*: *expand-inv* φ *n-ns*
shows *expand*$_T$ *n-ns* = *expand n-ns*
⟨*proof*⟩

lemma *expand-nofail*:
assumes *inv*:*expand-inv* φ *n-ns*
shows *nofail* (*expand*$_T$ *n-ns*)
⟨*proof*⟩

lemma [*intro*!]: *expand-inv* φ (
(‖
name = *expand-new-name expand-init*,
incoming = {*expand-init*},
new = {φ},
old = {},
next = {} ‖),
{})
⟨*proof*⟩

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
⟨proof⟩


lemma create-graph-finite: create-graph φ ≤ SPEC finite
⟨proof⟩

lemma create-graph-nofail: nofail (create-graph φ)
⟨proof⟩


abbreviation
  create-graph-rslt-exist ξ nds
  ≡ ∃ nd∈nds.
      expand-init∈incoming nd
    ∧ (∀ ψ∈old nd. ξ ⊨_n ψ) ∧ (∀ ψ∈next nd. ξ ⊨_n X_n ψ)
    ∧ {η. ∃ μ. μ U_n η ∈ old nd ∧ ξ ⊨_n η} ⊆ old nd

lemma L4-7:
  assumes ξ ⊨_n φ
   shows create-graph φ ≤ SPEC (create-graph-rslt-exist ξ)
  ⟨proof⟩


lemma expand-incoming-name-exist:
  assumes name (fst n-ns) > expand-init
  ∧ (∀ nm∈incoming (fst n-ns). nm ≠ expand-init ⟶ nm∈name ‘ (snd n-ns))
  ∧ expand-assm-incoming n-ns ∧ expand-name-ident (snd n-ns) (is ?Q n-ns)
```

and $\forall\, nd{\in}snd\ n\text{-}ns.$
$name\ nd > expand\text{-}init$
$\wedge\ (\forall\, nm{\in}incoming\ nd.\ nm \neq expand\text{-}init \longrightarrow nm{\in}name\ `\ (snd\ n\text{-}ns))$
(**is** $?P\ (snd\ n\text{-}ns)$)
shows $expand\ n\text{-}ns \leq SPEC\ (\lambda nm\text{-}nds.\ ?P\ (snd\ nm\text{-}nds))$
⟨*proof*⟩

lemma *create-graph--incoming-name-exist*:
shows $create\text{-}graph\ \varphi \leq SPEC\ (\lambda nds.\ \forall\, nd{\in}nds.\ expand\text{-}init < name\ nd \wedge (\forall\, nm{\in}incoming\ nd.\ nm \neq expand\text{-}init \longrightarrow nm{\in}name\ `\ nds))$
⟨*proof*⟩

abbreviation
$expand\text{-}rslt\text{-}all\text{--}ex\text{-}equiv\ \xi\ nd\ nds \equiv$
$(\exists\, nd'{\in}nds.$
$name\ nd \in incoming\ nd'$
$\wedge\ (\forall\, \psi{\in}old\ nd'.\ suffix\ 1\ \xi \models_n \psi) \wedge (\forall\, \psi{\in}next\ nd'.\ suffix\ 1\ \xi \models_n X_n\ \psi)$
$\wedge\ \{\eta.\ \exists\, \mu.\ \mu\ U_n\ \eta \in old\ nd' \wedge suffix\ 1\ \xi \models_n \eta\} \subseteq old\ nd')$

abbreviation
$expand\text{-}rslt\text{-}all\ \xi\ n\text{-}ns\ nm\text{-}nds \equiv$
$(\forall\, nd{\in}snd\ nm\text{-}nds.\ name\ nd \notin name\ `\ (snd\ n\text{-}ns) \wedge$
$(\forall\, \psi{\in}old\ nd.\ \xi \models_n \psi) \wedge (\forall\, \psi{\in}next\ nd.\ \xi \models_n X_n\ \psi)$
$\longrightarrow expand\text{-}rslt\text{-}all\text{--}ex\text{-}equiv\ \xi\ nd\ (snd\ nm\text{-}nds))$

lemma *expand-prop-all*:
assumes $expand\text{-}assm\text{-}incoming\ n\text{-}ns \wedge expand\text{-}name\text{-}ident\ (snd\ n\text{-}ns)$ (**is** $?Q\ n\text{-}ns$)
shows $expand\ n\text{-}ns \leq SPEC\ (expand\text{-}rslt\text{-}all\ \xi\ n\text{-}ns)$
(**is** $- \leq SPEC\ (?P\ n\text{-}ns)$)
⟨*proof*⟩

abbreviation
$create\text{-}graph\text{-}rslt\text{-}all\ \xi\ nds$
$\equiv \forall\, q{\in}nds.\ (\forall\, \psi{\in}old\ q.\ \xi \models_n \psi) \wedge (\forall\, \psi{\in}next\ q.\ \xi \models_n X_n\ \psi)$
$\longrightarrow (\exists\, q'{\in}nds.\ name\ q \in incoming\ q'$
$\wedge\ (\forall\, \psi{\in}old\ q'.\ suffix\ 1\ \xi \models_n \psi)$
$\wedge\ (\forall\, \psi{\in}next\ q'.\ suffix\ 1\ \xi \models_n X_n\ \psi)$
$\wedge\ \{\eta.\ \exists\, \mu.\ \mu\ U_n\ \eta \in old\ q' \wedge suffix\ 1\ \xi \models_n \eta\} \subseteq old\ q')$

lemma *L4-5*:
shows $create\text{-}graph\ \varphi \leq SPEC\ (create\text{-}graph\text{-}rslt\text{-}all\ \xi)$
⟨*proof*⟩

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 U_n η∈*old q* ⟶ η∈*old q*}|μ η. μ U_n η ∈ *subfrmlsn* φ},
gba-L = λ*q l*. *q*∈*qs* ∧ {*p*. *prop*$_n$(*p*)∈*old q*}⊆*l* ∧ {*p*. *nprop*$_n$(*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*
⟨*proof*⟩

lemma *create-gba-from-nodes--invar*: *gba* (*create-gba-from-nodes* φ *qs*)
⟨*proof*⟩

sublocale *gba*!: *gba create-gba-from-nodes* φ *qs* ⟨*proof*⟩

lemma *create-gba-from-nodes--fin*: *finite* (*g-V* (*create-gba-from-nodes* φ *qs*))
⟨*proof*⟩

lemma *create-gba-from-nodes--ipath*:
shows *ipath gba.E r*
⟷ (∀ *i*. *r i* ∈*qs* ∧ *name* (*r i*)∈*incoming* (*r* (*Suc i*)))
⟨*proof*⟩

lemma *create-gba-from-nodes--is-run*:
shows *gba.is-run r*
⟷ *expand-init* ∈ *incoming* (*r 0*)
∧ (∀ *i*. *r i*∈*qs* ∧ *name* (*r i*)∈*incoming* (*r* (*Suc i*)))
⟨*proof*⟩

context begin interpretation *LTL-Syntax* ⟨*proof*⟩

abbreviation
auto-run-j j ξ *q* ≡
(∀ ψ∈*old q*. *suffix j* ξ ⊨$_n$ ψ) ∧ (∀ ψ∈*next q*. *suffix j* ξ ⊨$_n$ X_n ψ) ∧
{η. ∃ μ. μ U_n η ∈ *old q* ∧ *suffix j* ξ ⊨$_n$ η} ⊆ *old q*

fun *auto-run* :: [*'a interprt*, *'a node set*] ⇒ *'a node word*
where

auto-run ξ *nds 0*
= (*SOME q*. $q \in nds \wedge$ *expand-init* $\in$ *incoming q* $\wedge$ *auto-run-j 0* ξ *q*)
| *auto-run* ξ *nds* (*Suc k*)
= (*SOME q'*. $q' \in nds \wedge$ *name* (*auto-run* ξ *nds k*) $\in$ *incoming q'*
$\wedge$ *auto-run-j* (*Suc k*) ξ *q'*)

lemma *run-propag-on-create-graph*:
assumes *ipath gba.E* σ
shows σ *k*$\in$*qs* $\wedge$ *name* (σ *k*)$\in$*incoming* (σ (*k*+*1*))
⟨*proof*⟩

lemma *expand-false-propag*:
assumes $false_n \notin$ *old* (*fst n-ns*) $\wedge$ ($\forall$ *nd*$\in$*snd n-ns*. $false_n \notin$ *old nd*)
(**is** *?Q n-ns*)
shows *expand n-ns* $\leq$ *SPEC* (λ*nm-nds*. $\forall$ *nd*$\in$*snd nm-nds*. $false_n \notin$ *old nd*)
⟨*proof*⟩

lemma *false-propag-on-create-graph*:
shows *create-graph* $\varphi \leq$ *SPEC* (λ*nds*. $\forall$ *nd*$\in$*nds*. $false_n \notin$ *old nd*)
⟨*proof*⟩

lemma *expand-and-propag*:
assumes μ and_n $\eta \in$ *old* (*fst n-ns*)
$\longrightarrow \{\mu, \eta\} \subseteq$ *old* (*fst n-ns*) $\cup$ *new* (*fst n-ns*) (**is** *?Q n-ns*)
and $\forall$ *nd*$\in$*snd n-ns*. μ and_n $\eta \in$ *old nd* $\longrightarrow \{\mu, \eta\} \subseteq$ *old nd* (**is** *?P* (*snd n-ns*))
shows *expand n-ns* $\leq$ *SPEC* (λ*nm-nds*. *?P* (*snd nm-nds*))
⟨*proof*⟩

lemma *and-propag-on-create-graph*:
shows *create-graph* φ
$\leq$ *SPEC* (λ*nds*. $\forall$ *nd*$\in$*nds*. μ and_n $\eta \in$ *old nd* $\longrightarrow \{\mu, \eta\} \subseteq$ *old nd*)
⟨*proof*⟩

lemma *expand-or-propag*:
assumes μ or_n $\eta \in$ *old* (*fst n-ns*)
$\longrightarrow \{\mu, \eta\} \cap$ (*old* (*fst n-ns*) $\cup$ *new* (*fst n-ns*)) $\neq \{\}$ (**is** *?Q n-ns*)
and $\forall$ *nd*$\in$*snd n-ns*. μ or_n $\eta \in$ *old nd* $\longrightarrow \{\mu, \eta\} \cap$ *old nd* $\neq \{\}$
(**is** *?P* (*snd n-ns*))
shows *expand n-ns* $\leq$ *SPEC* (λ*nm-nds*. *?P* (*snd nm-nds*))
⟨*proof*⟩

lemma *or-propag-on-create-graph*:
shows *create-graph* φ
$\leq$ *SPEC* (λ*nds*. $\forall$ *nd*$\in$*nds*. μ or_n $\eta \in$ *old nd* $\longrightarrow \{\mu, \eta\} \cap$ *old nd* $\neq \{\}$)

⟨*proof*⟩

abbreviation
next-propag--assm μ *n-ns* ≡
(*X*$_n$ μ ∈ *old* (*fst n-ns*) ⟶ μ ∈ *next* (*fst n-ns*))
∧ (∀ *nd*∈*snd n-ns*. *X*$_n$ μ ∈ *old nd* ∧ *name nd* ∈ *incoming* (*fst n-ns*)
⟶ μ ∈ *old* (*fst n-ns*) ∪ *new* (*fst n-ns*))

abbreviation
next-propag--rslt μ *ns* ≡
∀ *nd*∈*ns*. ∀ *nd'*∈*ns*. *X*$_n$ μ ∈ *old nd* ∧ *name nd* ∈ *incoming nd'* ⟶ μ ∈ *old nd'*

lemma *expand-next-propag*:
fixes *n-ns* :: - × *'a node set*
assumes *next-propag--assm* μ *n-ns*
∧ *next-propag--rslt* μ (*snd n-ns*)
∧ *expand-assm-incoming n-ns*
∧ *expand-name-ident* (*snd n-ns*) (**is** *?Q n-ns*)
shows *expand n-ns* ≤ *SPEC* (λ*r*. *next-propag--rslt* μ (*snd r*))
(**is** - ≤ *SPEC ?P*)
⟨*proof*⟩

lemma *next-propag-on-create-graph*:
shows *create-graph* φ
≤ *SPEC* (λ*nds*. ∀ *n*∈*nds*. ∀ *n'*∈*nds*. *X*$_n$ μ∈*old n* ∧ *name n*∈*incoming n'* ⟶ μ∈*old n'*)
⟨*proof*⟩

abbreviation
release-propag--assm μ η *n-ns* ≡
(μ *V*$_n$ η ∈ *old* (*fst n-ns*)
⟶ {μ, η}⊆*old* (*fst n-ns*)∪*new* (*fst n-ns*) ∨
(η∈*old* (*fst n-ns*)∪*new* (*fst n-ns*)) ∧ μ *V*$_n$ η ∈ *next* (*fst n-ns*))
∧ (∀ *nd*∈*snd n-ns*.
μ *V*$_n$ η ∈ *old nd* ∧ *name nd* ∈ *incoming* (*fst n-ns*)
⟶ {μ, η}⊆*old nd* ∨
(η∈*old nd* ∧ μ *V*$_n$ η ∈ *old* (*fst n-ns*)∪*new* (*fst n-ns*)))

abbreviation
release-propag--rslt μ η *ns* ≡
∀ *nd*∈*ns*.
∀ *nd'*∈*ns*.
μ *V*$_n$ η ∈ *old nd* ∧ *name nd* ∈ *incoming nd'*
⟶ {μ, η}⊆*old nd* ∨
(η∈*old nd* ∧ μ *V*$_n$ η ∈ *old nd'*)

lemma *expand-release-propag*:
fixes *n-ns* :: - × *'a node set*
assumes *release-propag--assm* μ η *n-ns*
$\wedge$ *release-propag--rslt* μ η (*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. *release-propag--rslt* μ η (*snd r*))
(**is** - $\leq$ *SPEC ?P*)
⟨*proof*⟩

lemma *release-propag-on-create-graph*:
shows *create-graph* φ
$\leq$ *SPEC* (λ*nds*. $\forall$ *n*∈*nds*. $\forall$ *n'*∈*nds*. μ V_n η∈*old n* $\wedge$ *name n*∈*incoming n'*
$\longrightarrow$ ({μ, η}⊆*old n* $\vee$ η∈*old n* $\wedge$ μ V_n η∈*old n'*))
⟨*proof*⟩

abbreviation
until-propag--assm f g n-ns ≡
(*f* U_n *g* ∈ *old* (*fst n-ns*)
$\longrightarrow$ (*g*∈*old* (*fst n-ns*)∪*new* (*fst n-ns*)
$\vee$ (*f*∈*old* (*fst n-ns*)∪*new* (*fst n-ns*) $\wedge$ *f* U_n *g* ∈ *next* (*fst n-ns*))))
$\wedge$ ($\forall$ *nd*∈*snd n-ns*. *f* U_n *g* ∈ *old nd* $\wedge$ *name nd* ∈ *incoming* (*fst n-ns*)
$\longrightarrow$ (*g*∈*old nd* $\vee$ (*f*∈*old nd* $\wedge$ *f* U_n *g* ∈*old* (*fst n-ns*)∪*new* (*fst n-ns*))))

abbreviation
until-propag--rslt f g ns ≡
$\forall$ *n*∈*ns*. $\forall$ *nd*∈*ns*. *f* U_n *g* ∈ *old n* $\wedge$ *name n* ∈ *incoming nd*
$\longrightarrow$ (*g* ∈ *old n* $\vee$ (*f*∈*old n* $\wedge$ *f* U_n *g* ∈ *old nd*))

lemma *expand-until-propag*:
fixes *n-ns* :: - × *'a node set*
assumes *until-propag--assm* μ η *n-ns*
$\wedge$ *until-propag--rslt* μ η (*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* μ η (*snd r*))
(**is** - $\leq$ *SPEC ?P*)
⟨*proof*⟩

lemma *until-propag-on-create-graph*:
shows *create-graph* φ
$\leq$ *SPEC* (λ*nds*. $\forall$ *n*∈*nds*. $\forall$ *n'*∈*nds*. μ U_n η∈*old n* $\wedge$ *name n*∈*incoming n'*
$\longrightarrow$ (η∈*old n* $\vee$ μ∈*old n* $\wedge$ μ U_n η∈*old n'*))
⟨*proof*⟩

definition *all-subfrmls* :: *'a node* ⇒ *'a frml set*
where

all-subfrmls n $\equiv \bigcup$ (*subfrmlsn* ‘ (*new n* $\cup$ *old n* $\cup$ *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*⟩

lemma *expand-all-subfrmls-propag*:
assumes *all-subfrmls* (*fst n-ns*) $\subseteq$ *B*
$\wedge$ ($\forall$ *nd*$\in$*snd n-ns*. *all-subfrmls nd* $\subseteq$ *B*) (**is** *?Q n-ns*)
shows *expand n-ns* $\leq$ *SPEC* ($\lambda r.$ $\forall$ *nd*$\in$*snd r*. *all-subfrmls nd* $\subseteq$ *B*)
(**is** - $\leq$ *SPEC ?P*)
⟨*proof*⟩

lemma *old-propag-on-create-graph*:
shows *create-graph* φ $\leq$ *SPEC* ($\lambda nds.$ $\forall$ *n*$\in$*nds*. *old n* $\subseteq$ *subfrmlsn* φ)
⟨*proof*⟩

lemma *L4-2--aux*:
assumes *run*: *ipath gba.E* σ
and μ U_n η $\in$ *old* (σ *0*)
and $\forall j.$ ($\forall i{<}j.$ {μ, μ U_n η} $\subseteq$ *old* (σ *i*)) $\longrightarrow$ η $\notin$ *old* (σ *j*)
shows $\forall i.$ {μ, μ U_n η} $\subseteq$ *old* (σ *i*) $\wedge$ η $\notin$ *old* (σ *i*)
⟨*proof*⟩

lemma *L4-2a*:
assumes *ipath gba.E* σ
and μ U_n η $\in$ *old* (σ *0*)
shows ($\forall i.$ {μ, μ U_n η} $\subseteq$ *old* (σ *i*) $\wedge$ η $\notin$ *old* (σ *i*))
$\vee$ ($\exists j.$ ($\forall i{<}j.$ {μ, μ U_n η} $\subseteq$ *old* (σ *i*)) $\wedge$ η $\in$ *old* (σ *j*))
(**is** *?A* $\vee$ *?B*)
⟨*proof*⟩

lemma *L4-2b*:
assumes *run*: *ipath gba.E* σ
and μ U_n η $\in$ *old* (σ *0*)
and *ACC*: *gba.is-acc* σ
shows $\exists j.$ ($\forall i{<}j.$ {μ, μ U_n η} $\subseteq$ *old* (σ *i*)) $\wedge$ η $\in$ *old* (σ *j*)
⟨*proof*⟩

lemma *L4-2c*:
assumes *run*: *ipath gba.E* σ
and μ V_n η $\in$ *old* (σ *0*)
shows $\forall i.$ η $\in$ *old* (σ *i*) $\vee$ ($\exists j{<}i.$ μ $\in$ *old* (σ *j*))
⟨*proof*⟩

lemma *L4-8′*:

assumes *ipath gba.E σ* (**is** *?inf-run σ*)
and *gba.is-acc σ* (**is** *?gbarel-accept σ*)
and $\forall i.$ *gba.L* (*σ i*) (*ξ i*) (**is** *?lgbarel-accept ξ σ*)
and $\psi \in$ *old* (*σ 0*)
shows $\xi \models_n \psi$
⟨*proof*⟩

lemma *L4-8*:
assumes *gba.is-acc-run σ*
and $\forall i.$ *gba.L* (*σ i*) (*ξ i*)
and $\psi \in$ *old* (*σ 0*)
shows $\xi \models_n \psi$
⟨*proof*⟩

lemma *expand-expand-init-propag*:
assumes ($\forall$ *nm*∈*incoming n′*. *nm* < *name n′*)
$\wedge$ *name n′* ≤ *name* (*fst n-ns*)
$\wedge$ (*incoming n′* ∩ *incoming* (*fst n-ns*) ≠ {}
$\longrightarrow$ *new n′* ⊆ *old* (*fst n-ns*) ∪ *new* (*fst n-ns*))
(**is** *?Q n-ns*)
and $\forall$ *nd*∈*snd n-ns*. $\forall$ *nm*∈*incoming n′*. *nm*∈*incoming nd* $\longrightarrow$ *new n′* ⊆ *old nd*
(**is** *?P* (*snd n-ns*))
shows *expand n-ns* ≤ *SPEC* (λ*r*. *name n′*≤*fst r* $\wedge$ *?P* (*snd r*))
⟨*proof*⟩

lemma *expand-init-propag-on-create-graph*:
shows *create-graph φ*
≤ *SPEC*(λ*nds*. $\forall$ *nd*∈*nds*. *expand-init*∈*incoming nd* $\longrightarrow$ *φ* ∈ *old nd*)
⟨*proof*⟩

lemma *L4-6*:
assumes *q*∈*gba.V0*
shows *φ*∈*old q*
⟨*proof*⟩

lemma *L4-9*:
assumes *acc*: *gba.accept ξ*
shows $\xi \models_n \varphi$
⟨*proof*⟩

lemma *L4-10*:
assumes $\xi \models_n \varphi$
shows *gba.accept ξ*
⟨*proof*⟩

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')
⟨*proof*⟩

corollary *create-graph--name-inj*:
shows *create-graph* $\varphi \leq$ *SPEC* (λ*nds*. *inj-on name nds*)
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *create-gba--invar*:
create-gba $\varphi \leq$ *SPEC gba*
⟨*proof*⟩

lemma *create-gba-acc*:
shows *create-gba* $\varphi \leq$ *SPEC*($\lambda\mathcal{A}$. $\forall\,\xi$. *gba.accept* $\mathcal{A}$ $\xi \longleftrightarrow \xi \models_n \varphi$)
⟨*proof*⟩

lemma *create-gba--name-inj*:
shows *create-gba* $\varphi \leq$ *SPEC*($\lambda\mathcal{A}$. (*inj-on name* (*g-V* $\mathcal{A}$)))
⟨*proof*⟩

lemma *create-gba--fin*:
shows *create-gba* $\varphi \leq$ *SPEC*($\lambda\mathcal{A}$. (*finite* (*g-V* $\mathcal{A}$)))
⟨*proof*⟩

lemma *create-graph-old-finite*:
create-graph $\varphi \leq$ *SPEC* (λ*nds*. $\forall\, nd{\in}$*nds*. *finite* (*old nd*))
⟨*proof*⟩

lemma *create-gba--old-fin*:
shows *create-gba* $\varphi \leq$ *SPEC*($\lambda\mathcal{A}$. $\forall\, nd{\in}$*g-V* $\mathcal{A}$. *finite* (*old nd*))
⟨*proof*⟩

lemma *create-gba--incoming-exists*:
shows *create-gba* φ
$\leq$ *SPEC*($\lambda\mathcal{A}$. $\forall\, nd{\in}$*g-V* $\mathcal{A}$. *incoming nd* $\subseteq$ *insert expand-init* (*name* ‘ (*g-V* $\mathcal{A}$)))
⟨*proof*⟩

lemma *create-gba--no-init*:
shows *create-gba* $\varphi \leq$ *SPEC*($\lambda\mathcal{A}$. *expand-init* $\notin$ *name* ‘ (*g-V* $\mathcal{A}$))

⟨*proof*⟩

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$\mathcal{A}$. *nds-invars* (*g-V* $\mathcal{A}$))
  ⟨*proof*⟩

theorem *T4-1*:
  shows *create-gba* φ ≤ *SPEC*(
    λ$\mathcal{A}$. *gba* $\mathcal{A}$
    ∧ *finite* (*g-V* $\mathcal{A}$)
    ∧ (∀ ξ. *gba.accept* $\mathcal{A}$ ξ ⟷ ξ ⊨$_n$ φ)
    ∧ (*nds-invars* (*g-V* $\mathcal{A}$)))
  ⟨*proof*⟩

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$\mathcal{A}$. *gba* $\mathcal{A}$ ∧ *finite* (*g-V* $\mathcal{A}$) ∧ (∀ ξ. *gba.accept* $\mathcal{A}$ ξ ⟷ ξ ⊨$_n$ φ))
  ⟨*proof*⟩

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* φ ≤ *SPEC* (λ*G*.
  igba G ∧ *finite* (*g-V G*) ∧ (∀ ξ. *igba.accept G* ξ ⟷ ξ ⊨$_n$ φ))
  ⟨*proof*⟩

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* $G\ \xi \longleftrightarrow \xi \models_n \varphi))$
  ⟨*proof*⟩

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*) ∈ *ltln-rel R*
| (*LTLnFalse*,*LTLnFalse*) ∈ *ltln-rel R*
| (*a*,*a'*)∈*R* ⟹ (*LTLnProp a*,*LTLnProp a'*) ∈ *ltln-rel R*
| (*a*,*a'*)∈*R* ⟹ (*LTLnNProp a*,*LTLnNProp a'*) ∈ *ltln-rel R*
| ⟦(*a*,*a'*)∈*ltln-rel R*; (*b*,*b'*)∈*ltln-rel R*⟧
  ⟹ (*LTLnAnd a b*,*LTLnAnd a' b'*) ∈ *ltln-rel R*
| ⟦(*a*,*a'*)∈*ltln-rel R*; (*b*,*b'*)∈*ltln-rel R*⟧
  ⟹ (*LTLnOr a b*,*LTLnOr a' b'*) ∈ *ltln-rel R*
| ⟦(*a*,*a'*)∈*ltln-rel R*⟧ ⟹ (*LTLnNext a*,*LTLnNext a'*) ∈ *ltln-rel R*
| ⟦(*a*,*a'*)∈*ltln-rel R*; (*b*,*b'*)∈*ltln-rel R*⟧
  ⟹ (*LTLnUntil a b*,*LTLnUntil a' b'*) ∈ *ltln-rel R*
| ⟦(*a*,*a'*)∈*ltln-rel R*; (*b*,*b'*)∈*ltln-rel R*⟧
  ⟹ (*LTLnRelease a b*,*LTLnRelease a' b'*) ∈ *ltln-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*) ∈ *ltln-rel R*
  (*LTLnFalse*,*z*) ∈ *ltln-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*⟩

lemma *ltln-rel-id*[*relator-props*]: $[\![\ R = Id\ ]\!] \Longrightarrow \langle R\rangle ltln\text{-}rel = Id$
⟨*proof*⟩

lemma *ltln-rel-id-simp*[*simp*]: $\langle Id\rangle ltln\text{-}rel = Id$ ⟨*proof*⟩

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*]:
$(LTLnTrue,\ LTLnTrue) \in \langle R\rangle ltln\text{-}rel$
$(LTLnFalse,\ LTLnFalse) \in \langle R\rangle ltln\text{-}rel$
$(LTLnProp,\ LTLnProp) \in R \to \langle R\rangle ltln\text{-}rel$
$(LTLnNProp,\ LTLnNProp) \in R \to \langle R\rangle ltln\text{-}rel$
$(LTLnAnd,\ LTLnAnd) \in \langle R\rangle ltln\text{-}rel \to \langle R\rangle ltln\text{-}rel \to \langle R\rangle ltln\text{-}rel$
$(LTLnOr,\ LTLnOr) \in \langle R\rangle ltln\text{-}rel \to \langle R\rangle ltln\text{-}rel \to \langle R\rangle ltln\text{-}rel$
$(LTLnNext,\ LTLnNext) \in \langle R\rangle ltln\text{-}rel \to \langle R\rangle ltln\text{-}rel$
$(LTLnUntil,\ LTLnUntil) \in \langle R\rangle ltln\text{-}rel \to \langle R\rangle ltln\text{-}rel \to \langle R\rangle ltln\text{-}rel$
$(LTLnRelease,\ LTLnRelease) \in \langle R\rangle ltln\text{-}rel \to \langle R\rangle ltln\text{-}rel \to \langle R\rangle ltln\text{-}rel$
⟨*proof*⟩

lemma *case-ltln-param*[*param*, *autoref-rules*]:
$(case\text{-}ltln, case\text{-}ltln) \in Rv \to Rv \to (R \to Rv)$
$\to (R \to Rv)$
$\to (\langle R\rangle ltln\text{-}rel \to \langle R\rangle ltln\text{-}rel \to Rv)$
$\to (\langle R\rangle ltln\text{-}rel \to \langle R\rangle ltln\text{-}rel \to Rv)$
$\to (\langle R\rangle ltln\text{-}rel \to Rv)$
$\to (\langle R\rangle ltln\text{-}rel \to \langle R\rangle ltln\text{-}rel \to Rv)$
$\to (\langle R\rangle ltln\text{-}rel \to \langle R\rangle ltln\text{-}rel \to Rv) \to \langle R\rangle ltln\text{-}rel \to Rv$
⟨*proof*⟩

lemma *rec-ltln-param*[*param*, *autoref-rules*]:
$(rec\text{-}ltln, rec\text{-}ltln) \in Rv \to Rv \to (R \to Rv)$

$\to (R \to Rv)$
$\to (\langle R\rangle ltln\text{-}rel \to \langle R\rangle ltln\text{-}rel \to Rv \to Rv \to Rv)$
$\to (\langle R\rangle ltln\text{-}rel \to \langle R\rangle ltln\text{-}rel \to Rv \to Rv \to Rv)$
$\to (\langle R\rangle ltln\text{-}rel \to Rv \to Rv)$
$\to (\langle R\rangle ltln\text{-}rel \to \langle R\rangle ltln\text{-}rel \to Rv \to Rv \to Rv)$
$\to (\langle R\rangle ltln\text{-}rel \to \langle R\rangle ltln\text{-}rel \to Rv \to Rv \to Rv)$
$\to \langle R\rangle ltln\text{-}rel \to Rv$
⟨*proof*⟩

lemma *case-ltln-mono*[*refine-mono*]:
assumes $\varphi = LTLnTrue \Longrightarrow a \leq a'$
assumes $\varphi = LTLnFalse \Longrightarrow b \leq b'$
assumes $\bigwedge p.\ \varphi = LTLnProp\ p \Longrightarrow c\ p \leq c'\ p$
assumes $\bigwedge p.\ \varphi = LTLnNProp\ p \Longrightarrow d\ p \leq d'\ p$
assumes $\bigwedge \mu\ \nu.\ \varphi = LTLnAnd\ \mu\ \nu \Longrightarrow e\ \mu\ \nu \leq e'\ \mu\ \nu$
assumes $\bigwedge \mu\ \nu.\ \varphi = LTLnOr\ \mu\ \nu \Longrightarrow f\ \mu\ \nu \leq f'\ \mu\ \nu$
assumes $\bigwedge \mu.\ \varphi = LTLnNext\ \mu \Longrightarrow g\ \mu \leq g'\ \mu$
assumes $\bigwedge \mu\ \nu.\ \varphi = LTLnUntil\ \mu\ \nu \Longrightarrow h\ \mu\ \nu \leq h'\ \mu\ \nu$
assumes $\bigwedge \mu\ \nu.\ \varphi = LTLnRelease\ \mu\ \nu \Longrightarrow i\ \mu\ \nu \leq i'\ \mu\ \nu$
shows *case-ltln a b c d e f g h i* $\varphi \leq$ *case-ltln a' b' c' d' e' f' g' h' i'* φ
⟨*proof*⟩

primrec *ltln-eq* **where**
ltln-eq eq LTLnTrue f $\longleftrightarrow$ (*case f of LTLnTrue* $\Rightarrow$ *True* | - $\Rightarrow$ *False*)
| *ltln-eq eq LTLnFalse f* $\longleftrightarrow$ (*case f of LTLnFalse* $\Rightarrow$ *True* | - $\Rightarrow$ *False*)
| *ltln-eq eq* (*LTLnProp p*) *f* $\longleftrightarrow$ (*case f of LTLnProp p'* $\Rightarrow$ *eq p p'* | - $\Rightarrow$ *False*)
| *ltln-eq eq* (*LTLnNProp p*) *f* $\longleftrightarrow$ (*case f of LTLnNProp p'* $\Rightarrow$ *eq p p'* | - $\Rightarrow$ *False*)
| *ltln-eq eq* (*LTLnAnd* $\mu\ \nu$) *f*
$\longleftrightarrow$ (*case f of LTLnAnd* $\mu'\ \nu'$ $\Rightarrow$ *ltln-eq eq* $\mu\ \mu' \wedge$ *ltln-eq eq* $\nu\ \nu'$ | - $\Rightarrow$ *False*)
| *ltln-eq eq* (*LTLnOr* $\mu\ \nu$) *f*
$\longleftrightarrow$ (*case f of LTLnOr* $\mu'\ \nu'$ $\Rightarrow$ *ltln-eq eq* $\mu\ \mu' \wedge$ *ltln-eq eq* $\nu\ \nu'$ | - $\Rightarrow$ *False*)
| *ltln-eq eq* (*LTLnNext* φ) *f*
$\longleftrightarrow$ (*case f of LTLnNext* φ' $\Rightarrow$ *ltln-eq eq* $\varphi\ \varphi'$ | - $\Rightarrow$ *False*)
| *ltln-eq eq* (*LTLnUntil* $\mu\ \nu$) *f*
$\longleftrightarrow$ (*case f of LTLnUntil* $\mu'\ \nu'$ $\Rightarrow$ *ltln-eq eq* $\mu\ \mu' \wedge$ *ltln-eq eq* $\nu\ \nu'$ | - $\Rightarrow$ *False*)
| *ltln-eq eq* (*LTLnRelease* $\mu\ \nu$) *f*
$\longleftrightarrow$ (*case f of*
LTLnRelease $\mu'\ \nu'$ $\Rightarrow$ *ltln-eq eq* $\mu\ \mu' \wedge$ *ltln-eq eq* $\nu\ \nu'$
| - $\Rightarrow$ *False*)

lemma *ltln-eq-autoref*[*autoref-rules*]:
assumes *EQP*: (*eq*,*op*=) $\in R \to R \to$ *bool-rel*
shows (*ltln-eq eq*, *op*=) $\in \langle R\rangle ltln\text{-}rel \to \langle R\rangle ltln\text{-}rel \to$ *bool-rel*
⟨*proof*⟩

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* $\rightarrow \langle R \rangle$*ltln-rel* $\rightarrow$ *comp-res-rel*
⟨*proof*⟩

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

abbreviation (*input*) *node-name-rel* $\equiv$ *Id* :: (*node-name-impl*$\times$*node-name*) *set*

lemma *case-ltln-gtransfer*:
assumes
γ *ai* $\leq$ *a*
γ *bi* $\leq$ *b*
$\bigwedge a.\ \gamma$ (*ci a*) $\leq$ *c a*
$\bigwedge a.\ \gamma$ (*di a*) $\leq$ *d a*
$\bigwedge$*ltln1 ltln2*. γ (*ei ltln1 ltln2*) $\leq$ *e ltln1 ltln2*
$\bigwedge$*ltln1 ltln2*. γ (*fi ltln1 ltln2*) $\leq$ *f ltln1 ltln2*
$\bigwedge$*ltln*. γ (*gi ltln*) $\leq$ *g ltln*
$\bigwedge$*ltln1 ltln2*. γ (*hi ltln1 ltln2*) $\leq$ *h ltln1 ltln2*
$\bigwedge$*ltln1 ltln2*. γ (*ii ltln1 ltln2*) $\leq$ *i ltln1 ltln2*
shows γ (*case-ltln ai bi ci di ei fi gi hi ii* φ)
$\leq$ (*case-ltln a b c d e f g h i* φ)
⟨*proof*⟩

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-ltln ai bi ci di ei fi gi hi ii* $\varphi \neq$ *dSUCCEED*
⟨*proof*⟩

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*

} ⟨*proof*⟩

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

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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
⟨proof⟩
```

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 ⟨proof⟩

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)
  ⟨proof⟩


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)
  ⟨proof⟩
```

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 $\leftarrow$ *create-graph*$_T$ φ;
  ASSERT (*nds-invars nds*);
  RETURN (*gba-rename-ext* (λ-. ()) *name* (*create-gba-from-nodes* φ *nds*))
  }
⟨*proof*⟩

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

definition *build-succ nds* =
  FOREACH
    nds (λq' *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*
  }))
  ⟨*proof*⟩

lemma *build-succ-aux2*:
  assumes *NINIT*: *expand-init* ∉ *name*‘*nds*
  assumes *CL*: ⋀*nd*. *nd*∈*nds* ⟹ *incoming nd* ⊆ *insert expand-init* (*name*‘*nds*)
  shows
  (λ*qn*. *dflt-None-set*
    {*qn'*. ∃ *q'*. *q'*∈*nds* ∧ *qn'* = *name q'* ∧ *qn*∈*incoming q'* ∧ *qn*≠*expand-init* })
  = (λ*qn*. *dflt-None-set* (*succ-of-E*
      (*rename-E name* {(*q*, *q'*). *q* ∈ *nds* ∧ *q'* ∈ *nds* ∧ *name q* ∈ *incoming q'*})
qn))
  (**is** (λ*qn*. *dflt-None-set* (*?L qn*)) = (λ*qn*. *dflt-None-set* (*?R qn*)))
  ⟨*proof*⟩

lemma *build-succ-correct*:
  assumes *NINIT*: *expand-init* ∉ *name*‘*nds*

assumes *FIN*: *finite nds*
assumes *CL*: $\bigwedge nd.\ nd \in nds \Longrightarrow$ *incoming nd* $\subseteq$ *insert expand-init* (*name*`*nds*)
shows *build-succ nds* $\leq$ *SPEC* (λr.
E-of-succ (λqn. *the-default* {} (*r qn*))
= *rename-E* (λu. *name u*) {(*q*, *q'*). *q* $\in$ *nds* $\wedge$ *q'* $\in$ *nds* $\wedge$ *name q* $\in$ *incoming q'*})
⟨*proof*⟩

context begin interpretation *LTL-Syntax* ⟨*proof*⟩
primrec *until-frmlsn* :: *'a frml* $\Rightarrow$ (*'a frml* $\times$ *'a frml*) *set* **where**
until-frmlsn (μ *and*$_n$ ψ) = (*until-frmlsn* μ) $\cup$ (*until-frmlsn* ψ)
| *until-frmlsn* (*X*$_n$ μ) = *until-frmlsn* μ
| *until-frmlsn* (μ *U*$_n$ ψ) = *insert* (μ, ψ) ((*until-frmlsn* μ) $\cup$ (*until-frmlsn* ψ))
| *until-frmlsn* (μ *V*$_n$ ψ) = (*until-frmlsn* μ) $\cup$ (*until-frmlsn* ψ)
| *until-frmlsn* (μ *or*$_n$ ψ) = (*until-frmlsn* μ) $\cup$ (*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 $\in$ *subfrmlsn* φ}
⟨*proof*⟩

definition *build-F nds* φ
$\equiv$ ($\lambda(\mu,\eta)$. *name* ` {*q* $\in$ *nds*. (*LTLnUntil* $\mu$ $\eta$ $\in$ *old q* $\longrightarrow$ $\eta$ $\in$ *old q*)}) `
until-frmlsn φ

lemma *build-F-correct*: *build-F nds* φ =
{*name* ` *A* |*A*. $\exists \mu\ \eta$. *A* = {*q* $\in$ *nds*. *LTLnUntil* μ η $\in$ *old q* $\longrightarrow$ η $\in$ *old q*} $\wedge$
LTLnUntil μ η $\in$ *subfrmlsn* φ}
⟨*proof*⟩

definition *pn-props ps* $\equiv$ *FOREACHi*
(λ*it* (*P*,*N*). *P* = {*p*. *LTLnProp p* $\in$ *ps* $-$ *it*} $\wedge$ *N* = {*p*. *LTLnNProp p* $\in$ *ps* $-$ *it*})
ps (λ*p* (*P*,*N*).
case p of LTLnProp p $\Rightarrow$ *RETURN* (*insert p P*,*N*)
| *LTLnNProp p* $\Rightarrow$ *RETURN* (*P*, *insert p N*)
| - $\Rightarrow$ *RETURN* (*P*,*N*)
) ({},{})

lemma *pn-props-correct*:
assumes [*simp*]: *finite ps*
shows *pn-props ps* $\leq$ *SPEC*($\lambda r.\ r =$
($\{p.\ LTLnProp\ p \in ps\}$, $\{p.\ LTLnNProp\ p \in ps\}$))
⟨*proof*⟩

definition *pn-map nds* $\equiv$ *FOREACH nds*
($\lambda nd\ m.$ *do* {
PN $\leftarrow$ *pn-props* (*old nd*);
RETURN (m(*name nd* $\mapsto$ *PN*))
}) *Map.empty*

lemma *pn-map-correct*:
assumes [*simp*]: *finite nds*
assumes *FIN'*: $\bigwedge nd.\ nd \in nds \Longrightarrow$ *finite* (*old nd*)
assumes *INJ*: *inj-on name nds*
shows *pn-map nds* $\leq$ *SPEC* ($\lambda r.\ \forall\, qn.$
case r qn of
None $\Rightarrow$ *qn* $\notin$ *name*`*nds*
| *Some* (*P*,*N*) $\Rightarrow$ *qn* $\in$ *name*`*nds*
$\wedge$ *P* = $\{p.\ LTLnProp\ p \in old\ (the\text{-}inv\text{-}into\ nds\ name\ qn)\}$
$\wedge$ *N* = $\{p.\ LTLnNProp\ p \in old\ (the\text{-}inv\text{-}into\ nds\ name\ qn)\}$
)
⟨*proof*⟩

definition *cr-rename-gba nds* $\varphi \equiv$ *do* {
let V = *name* ` *nds*;
let V0 = *name* ` $\{q \in nds.\ expand\text{-}init \in incoming\ q\}$;
succmap $\leftarrow$ *build-succ nds*;
let E = *E-of-succ* (*the-default* {} *o succmap*);
let F = *build-F nds* φ;
pnm $\leftarrow$ *pn-map nds*;
let L = ($\lambda qn\ l.$ *case pnm qn of*
None $\Rightarrow$ *False*
| *Some* (*P*,*N*) $\Rightarrow$ ($\forall\, p{\in}P.\ p{\in}(l{:::}_r\langle Id\rangle fun\text{-}set\text{-}rel)$) $\wedge$ ($\forall\, p{\in}N.\ p{\notin}l$)
);
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*)$\in$*Id* (φ',φ)$\in$*Id*
shows *cr-rename-gba nds'* φ'
$\leq \Downarrow$*Id* (*RETURN* (*gba-rename-ext* (λ-. ()) *name* (*create-gba-from-nodes* φ *nds*)))
⟨*proof*⟩

definition *create-name-gba-aimpl* $\varphi \equiv$ *do* {
  nds $\leftarrow$ *create-graph-aimpl* φ;
  ASSERT (*nds-invars nds*);
  cr-rename-gba nds φ
}

lemma *create-name-gba-aimpl-refine*:
  create-name-gba-aimpl $\varphi \leq \Downarrow Id$ (*create-name-gba* φ)
  ⟨*proof*⟩

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*
  ⟨*proof*⟩
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*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*

⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

schematic-lemma *pn-map-code-aux*: *RETURN ?c* ≤ *pn-map-impl x*
⟨*proof*⟩
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*)
⟨*proof*⟩
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*
⟨*proof*⟩

schematic-lemma *cr-rename-gba-code-aux*: *RETURN ?c* $\leq$ *cr-rename-gba-impl x y*
⟨*proof*⟩
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*}
⟨*proof*⟩

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*)
⟨*proof*⟩

context begin interpretation *autoref-syn* ⟨*proof*⟩
lemma *lsnds-empty-autoref*[*autoref-rules*]:
assumes *PREFER-id R*
shows ([],{})∈⟨*R*⟩*ls-nds-rel*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *ls-nds-image-autoref-aux*:
assumes [*autoref-rules*]: (*fi*,*f*) ∈ *Ra* → *Rb*
assumes (*l*,*s*) ∈ ⟨*Ra*⟩*ls-nds-rel*
assumes [*simp*]: ∀*x*. *name* (*f x*) = *name x*
shows (*map fi l*, *f‘s*) ∈ ⟨*Rb*⟩*ls-nds-rel*
⟨*proof*⟩

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 $(map\ fi,\ (OP\ op\ `\ ::: (Ra \to Rb) \to \langle Ra \rangle ls\text{-}nds\text{-}rel \to \langle Rb \rangle ls\text{-}nds\text{-}rel)\$f)$
$\in \langle Ra \rangle ls\text{-}nds\text{-}rel \to \langle Rb \rangle ls\text{-}nds\text{-}rel$
⟨*proof*⟩

lemma *list-set-autoref-to-list*[*autoref-ga-rules*]:
shows *is-set-to-sorted-list* $(\lambda\text{- -}.\ True)$ *R ls-nds-rel id*
⟨*proof*⟩

end

context begin interpretation *autoref-syn* ⟨*proof*⟩
lemma [*autoref-itype*]:
upd-incoming
$::_i \langle Im,\ I \rangle_i i\text{-}node \to_i \langle \langle Im',\ I \rangle_i i\text{-}node \rangle_i i\text{-}set \to_i \langle \langle Im',\ I \rangle_i i\text{-}node \rangle_i i\text{-}set$
⟨*proof*⟩
end

term *upd-incoming*
schematic-lemma *upd-incoming-impl-aux*:
assumes *REL-IS-ID R*
shows $(?c,\ upd\text{-}incoming) \in \langle Rm1,R \rangle node\text{-}rel$
$\to \langle \langle Rm2,R \rangle node\text{-}rel \rangle ls\text{-}nds\text{-}rel$
$\to \langle \langle Rm2,R \rangle node\text{-}rel \rangle ls\text{-}nds\text{-}rel$
⟨*proof*⟩

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*: $(?c,\ expand\text{-}aimpl) \in$
$\langle unit\text{-}rel,Id \rangle node\text{-}rel \times_r \langle \langle unit\text{-}rel,Id \rangle node\text{-}rel \rangle ls\text{-}nds\text{-}rel$
$\to \langle nat\text{-}rel \times_r \langle \langle unit\text{-}rel,Id \rangle node\text{-}rel \rangle ls\text{-}nds\text{-}rel \rangle nres\text{-}rel$
⟨*proof*⟩

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

context begin interpretation *autoref-syn* ⟨*proof*⟩
lemma [*autoref-itype*]: $expand_T$
$::_i \langle \langle i\text{-}unit,\ I \rangle_i i\text{-}node,\ \langle \langle i\text{-}unit,\ I \rangle_i i\text{-}node \rangle_i i\text{-}set \rangle_i i\text{-}prod$
$\to_i \langle \langle i\text{-}nat, \langle \langle i\text{-}unit,\ I \rangle_i i\text{-}node \rangle_i i\text{-}set \rangle_i i\text{-}prod \rangle_i i\text{-}nres\ \langle proof \rangle$

lemma *expand-autoref*[*autoref-rules*]:
assumes *ID*: *PREFER-id R*
assumes *A*: $(n\text{-}ns',\ n\text{-}ns)$
$\in \langle unit\text{-}rel,R \rangle node\text{-}rel \times_r \langle \langle unit\text{-}rel,R \rangle node\text{-}rel \rangle list\text{-}set\text{-}rel$

assumes *B*: *SIDE-PRECOND* (
let (*n*,*ns*)=*n-ns* *in* *inj-on* *name* *ns* ∧ (∀ *n'*∈*ns*. *name* *n* > *name* *n'*)
)
shows (*expand-impl* *n-ns'*,
(*OP* *expand-aimpl*
::: ⟨*unit-rel*,*R*⟩*node-rel* ×$_r$ ⟨⟨*unit-rel*,*R*⟩*node-rel*⟩*list-set-rel*
→ ⟨*nat-rel* ×$_r$ ⟨⟨*unit-rel*,*R*⟩*node-rel*⟩*list-set-rel*⟩*nres-rel*)$*n-ns*)
∈ ⟨*nat-rel* ×$_r$ ⟨⟨*unit-rel*,*R*⟩*node-rel*⟩*list-set-rel*⟩*nres-rel*
⟨*proof*⟩

end

schematic-lemma *expand-code-aux*: *RETURN* *?c* ≤ *expand-impl* *n-ns*
⟨*proof*⟩
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*)
∈ ⟨*R*⟩*ltln-rel* → ⟨⟨⟨*unit-rel*,*R*⟩*node-rel*⟩*list-set-rel*⟩*nres-rel*
⟨*proof*⟩
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*
⟨*proof*⟩
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* ⇒ -))
∈ ⟨*Id*⟩*ltln-rel* → (*?R*::(*?'c*×-) *set*)
⟨*proof*⟩
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*)
∈ ⟨*R*⟩*ltln-rel* → ⟨*gbav-impl-rel-ext* *unit-rel* *nat-rel* (⟨*R*⟩*fun-set-rel*)⟩*nres-rel*
(**is** -:-→⟨*?R*⟩*nres-rel*)
⟨*proof*⟩

schematic-lemma *create-name-gba-code-aux*: *RETURN ?c* ≤ *create-name-gba-impl* φ
⟨*proof*⟩
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*⟩
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*
⟨*proof*⟩
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
⟨*proof*⟩
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.