Converting Linear-Time Temporal Logic to Generalized Büchi Automata

Alexander Schimpf and Peter Lammich

May 26, 2024

Abstract

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

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

Contents

1 Introduction

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

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

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

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

2 LTL to GBA translation

theory *LTL-to-GBA*
imports
CAVA-Base.CAVA-Base
LTL.LTL
CAVA-Automata.Automata
begin

2.1 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*⟩

2.2 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* (λ*r'*. *g r'* ≤ *SPEC P*)
⟨*proof*⟩

lemma *SPEC-rule-param2*:
assumes *f x* ≤ *SPEC* (*P x*)
and ⋀*r1 r2*. (*P x*) (*r1*, *r2*) ⟹ (*P x'*) (*r1*, *r2*)
shows *f x* ≤ *SPEC* (*P x'*)
⟨*proof*⟩

lemma *SPEC-rule-weak*:
assumes *f x* ≤ *SPEC* (*Q x*) **and** *f x* ≤ *SPEC* (*P x*)
and ⋀*r1 r2*. ⟦(*Q x*) (*r1*, *r2*); (*P x*) (*r1*, *r2*)⟧ ⟹ (*P x'*) (*r1*, *r2*)
shows *f x* ≤ *SPEC* (*P x'*)
⟨*proof*⟩

lemma *SPEC-rule-weak-nested2*: ⟦*f* ≤ *SPEC Q*; *f* ≤ *SPEC P*;
⋀*r1 r2*. ⟦*Q* (*r1*, *r2*); *P* (*r1*, *r2*)⟧ ⟹ *g* (*r1*, *r2*) ≤ *SPEC P*⟧
⟹ *f* ≤ *SPEC* (λ*r'*. *g r'* ≤ *SPEC P*)
⟨*proof*⟩

2.3 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 ltlr*

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

fun *new1* **where**
  new1 (μ and_r ψ) = {μ,ψ}
| *new1* (μ U_r ψ) = {μ}
| *new1* (μ R_r ψ) = {ψ}
| *new1* (μ or_r ψ) = {μ}
| *new1* - = {}

fun *next1* **where**
  next1 (X_r ψ) = {ψ}
| *next1* (μ U_r ψ) = {μ U_r ψ}
| *next1* (μ R_r ψ) = {μ R_r ψ}
| *next1* - = {}

fun *new2* **where**
  new2 (μ U_r ψ) = {ψ}
| *new2* (μ R_r ψ) = {μ, ψ}
| *new2* (μ or_r ψ) = {ψ}
| *new2* - = {}

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

lemma *expand-new-name-expand-init*: *expand-init* < *expand-new-name nm*
  ⟨*proof*⟩

lemma *expand-new-name-step*[*intro*]:

fixes $n :: 'a\ node$
shows $name\ n < expand\text{-}new\text{-}name\ (name\ n)$
⟨*proof*⟩

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

abbreviation
$upd\text{-}incoming\text{-}f\ n \equiv (\lambda n'.$
$\quad if\ (old\ n' = old\ n \wedge next\ n' = next\ n)\ then$
$\quad\quad n'(\!|\ incoming := incoming\ n \cup incoming\ n'\ |\!)$
$\quad else\ n'$
$)$

definition
$upd\text{-}incoming\ n\ ns \equiv ((upd\text{-}incoming\text{-}f\ n)\ `\ ns)$

lemma *upd-incoming--elem*:
assumes $nd \in upd\text{-}incoming\ n\ ns$
shows $nd \in ns$
$\quad \vee\ (\exists\, nd' \in ns.\ nd = nd'(\!|\ incoming := incoming\ n \cup incoming\ nd'\ |\!) \wedge$
$\quad\quad old\ nd' = old\ n \wedge$
$\quad\quad next\ nd' = next\ n)$
⟨*proof*⟩

lemma *upd-incoming--ident-node*:
assumes $nd \in upd\text{-}incoming\ n\ ns$ **and** $nd \in ns$
shows $incoming\ n \subseteq incoming\ nd \vee \neg\ (old\ nd = old\ n \wedge next\ nd = next\ n)$
⟨*proof*⟩

lemma *upd-incoming--ident*:
assumes $\forall\, n \in ns.\ P\ n$
and $\bigwedge n.\ [\![n \in ns;\ P\ n]\!] \Longrightarrow (\bigwedge v.\ P\ (n(\!|\ incoming := v\ |\!)))$
shows $\forall\, n \in upd\text{-}incoming\ n\ ns.\ P\ n$
⟨*proof*⟩

lemma *name-upd-incoming-f*[*simp*]: $name\ (upd\text{-}incoming\text{-}f\ n\ x) = name\ x$
⟨*proof*⟩

lemma *name-upd-incoming*[*simp*]:
$name\ `\ (upd\text{-}incoming\ n\ ns) = name\ `\ ns$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

abbreviation *expand-body*
where
$expand\text{-}body \equiv (\lambda expand\ (n,ns).$

```
    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 (λx. x∈(new n));
      let n = n⦇ new := new n - {φ} ⦈;
      if (∃ q. φ = prop_r(q) ∨ φ = nprop_r(q)) then
        (if (not_r φ) ∈ old n then RETURN (name n, ns)
         else expand (n⦇ old := {φ} ∪ old n ⦈, ns))
      else if φ = true_r then expand (n⦇ old := {φ} ∪ old n ⦈, ns)
      else if φ = false_r then RETURN (name n, ns)
      else if (∃ ν μ. (φ = ν and_r μ) ∨ (φ = X_r 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 $I0$: $\Phi\ x$
assumes IS: $\bigwedge f\ x.\ [\![\ \bigwedge x.\ \Phi\ x \Longrightarrow f\ x \leq M\ x;\ \Phi\ x;\ f \leq REC\ body\ ]\!]$
$\Longrightarrow$ *body f x* $\leq M\ x$
shows $REC\ body\ x \leq M\ x$
⟨*proof*⟩

lemma *expand-rec-rule*:
assumes $I0$: $\Phi\ x$
assumes IS: $\bigwedge f\ x.\ [\![\ \bigwedge x.\ f\ x \leq expand\ x;\ \bigwedge x.\ \Phi\ x \Longrightarrow f\ x \leq M\ x;\ \Phi\ x\ ]\!]$
$\Longrightarrow$ *expand-body f x* $\leq M\ x$
shows *expand x* $\leq M\ x$
⟨*proof*⟩

abbreviation
expand-assm-incoming n-ns
$\equiv (\forall\, nm{\in}incoming\ (fst\ n\text{-}ns).\ name\ (fst\ n\text{-}ns) > nm)$
$\wedge\ 0 < name\ (fst\ n\text{-}ns)$
$\wedge\ (\forall\, q{\in}snd\ n\text{-}ns.$
$name\ (fst\ n\text{-}ns) > name\ q$
$\wedge\ (\forall\, nm{\in}incoming\ q.\ name\ (fst\ n\text{-}ns) > nm))$

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

abbreviation
expand-rslt-exist--node-prop ξ n nd
$\equiv incoming\ n \subseteq incoming\ nd$
$\wedge\ (\forall\, \psi{\in}old\ nd.\ \xi \models_r \psi) \wedge (\forall\, \psi{\in}next\ nd.\ \xi \models_r X_r\ \psi)$

$\wedge\ \{\eta.\ \exists\,\mu.\ \mu\ U_r\ \eta \in old\ nd \wedge \xi \models_r \eta\} \subseteq old\ nd$

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

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

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

lemma *expand-name-propag*:
assumes *expand-assm-incoming n-ns* $\wedge$ *expand-name-ident* (*snd n-ns*) (**is** *?Q n-ns*)
shows *expand n-ns* $\leq$ *SPEC* ($\lambda r.$ *expand-rslt-incoming r*
$\wedge$ *expand-rslt-name n-ns r*
$\wedge$ *expand-name-ident* (*snd r*))
(**is** *expand* - $\leq$ *SPEC* (*?P n-ns*))
⟨*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*:
expand n-ns $\leq$ *SPEC* ($\lambda r.$ *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* $\varphi \equiv$ *Pow* (*subfrmlsr* φ) $\times$ *Pow* (*subfrmlsr* φ)

lemma *old-next-limit-finite*: *finite* (*old-next-limit* φ)
⟨*proof*⟩

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

lemma *expand-ord-wf*[*simp*]: *wf* (*expand-ord* φ)
⟨*proof*⟩

abbreviation
expand-inv-node φ *n*
≡ *finite* (*new n*) ∧ *finite* (*old n*) ∧ *finite* (*next n*)
∧ (*new n*) ∪ (*old n*) ∪ (*next n*) ⊆ *subfrmlsr* φ

abbreviation
expand-inv-result φ *ns* ≡ *finite ns* ∧ (∀ *n*′∈*ns*. (*new n*′) ∪ (*old n*′) ∪ (*next n*′) ⊆ *subfrmlsr* φ)

definition
expand-inv φ *n-ns* ≡ (*case n-ns of* (*n*, *ns*) ⇒ *expand-inv-node* φ *n* ∧ *expand-inv-result* φ *ns*)

lemma *new1-less-sum*:
size-set (*new1* φ) < *size-set* {φ}
⟨*proof*⟩

lemma *new2-less-sum*:
size-set (*new2* φ) < *size-set* {φ}
⟨*proof*⟩

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

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

lemma *next1-finite*[*intro*]: *finite* (*next1* ψ)
⟨*proof*⟩
lemma *next1-subset-frmls*: $\varphi \in$ *next1* $\psi \Longrightarrow \varphi \in$ *subfrmlsr* ψ
⟨*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 − {ψ},
                old := {ψ} ∪ *old* n |)) ∨
        (n' = n(| *new* := *new1* ψ ∪ (*new* n − {ψ}),
                old := {ψ} ∪ *old* n,
                next := *next1* ψ ∪ *next* n |)) ∨
        (n' = n(| *name* := nm,
                new := *new2* ψ ∪ (*new* n − {ψ}),
                old := {ψ} ∪ *old* n |))
        (**is** *?case1* ∨ *?case2* ∨ *?case3*)
  shows ((n', ns'), (n, ns))∈*expand-ord* φ ∧ *expand-inv* φ (n', ns')
        (**is** *?concl1* ∧ *?concl2*)
⟨*proof*⟩

lemma *expand-term-prop-help*:
  assumes ((n', ns'), (n, ns))∈*expand-ord* φ ∧ *expand-inv* φ (n', ns')
    and *assm-rule*: ⟦*expand-inv* φ (n', ns'); ((n', ns'), n, ns) ∈ *expand-ord* φ⟧
      ⟹ f (n', ns') ≤ *SPEC* P
  shows f (n', ns') ≤ *SPEC* P
  ⟨*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*: *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 *inv*: *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* $\varphi \leq$ *SPEC finite*
⟨*proof*⟩

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

abbreviation
create-graph-rslt-exist ξ *nds*
$\equiv \exists\, nd{\in}nds.$
expand-init$\in$*incoming nd*
$\wedge\ (\forall\, \psi{\in}old\ nd.\ \xi \models_r \psi) \wedge (\forall\, \psi{\in}next\ nd.\ \xi \models_r X_r\ \psi)$
$\wedge\ \{\eta.\ \exists\, \mu.\ \mu\ U_r\ \eta \in old\ nd \wedge \xi \models_r \eta\} \subseteq old\ nd$

lemma *L4-7*:
assumes $\xi \models_r \varphi$
shows *create-graph* $\varphi \leq$ *SPEC* (*create-graph-rslt-exist* ξ)
⟨*proof*⟩

lemma *expand-incoming-name-exist*:
assumes *name* (*fst n-ns*) > *expand-init*
$\wedge\ (\forall\, nm{\in}incoming\ (fst\ n\text{-}ns).\ nm \neq expand\text{-}init \longrightarrow nm{\in}name\ `\ (snd\ n\text{-}ns))$
$\wedge$ *expand-assm-incoming n-ns* $\wedge$ *expand-name-ident* (*snd n-ns*) (**is** *?Q n-ns*)
and $\forall\, nd{\in}snd\ n\text{-}ns.$
name nd > *expand-init*
$\wedge\ (\forall\, nm{\in}incoming\ nd.\ nm \neq expand\text{-}init \longrightarrow nm{\in}name\ `\ (snd\ n\text{-}ns))$
(**is** *?P* (*snd n-ns*))
shows *expand n-ns* $\leq$ *SPEC* (λ*nm-nds. ?P* (*snd nm-nds*))
⟨*proof*⟩

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

abbreviation
expand-rslt-all--ex-equiv ξ *nd nds* $\equiv$
$(\exists\, nd'{\in}nds.$
name nd $\in$ *incoming nd*$'$
$\wedge\ (\forall\, \psi{\in}old\ nd'.\ suffix\ 1\ \xi \models_r \psi) \wedge (\forall\, \psi{\in}next\ nd'.\ suffix\ 1\ \xi \models_r X_r\ \psi)$
$\wedge\ \{\eta.\ \exists\, \mu.\ \mu\ U_r\ \eta \in old\ nd' \wedge suffix\ 1\ \xi \models_r \eta\} \subseteq old\ nd')$

abbreviation
expand-rslt-all ξ *n-ns nm-nds* $\equiv$
$(\forall\, nd{\in}snd\ nm\text{-}nds.\ name\ nd \notin name\ `\ (snd\ n\text{-}ns) \wedge$

$(\forall\,\psi{\in}old\ nd.\ \xi \models_r \psi) \wedge (\forall\,\psi{\in}next\ nd.\ \xi \models_r X_r\ \psi)$
$\longrightarrow$ *expand-rslt-all--ex-equiv* ξ *nd* (*snd nm-nds*))

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

abbreviation
create-graph-rslt-all ξ *nds*
$\equiv \forall\,q{\in}nds.\ (\forall\,\psi{\in}old\ q.\ \xi \models_r \psi) \wedge (\forall\,\psi{\in}next\ q.\ \xi \models_r X_r\ \psi)$
$\longrightarrow (\exists\,q'{\in}nds.\ name\ q \in incoming\ q'$
$\wedge\ (\forall\,\psi{\in}old\ q'.\ suffix\ 1\ \xi \models_r \psi)$
$\wedge\ (\forall\,\psi{\in}next\ q'.\ suffix\ 1\ \xi \models_r X_r\ \psi)$
$\wedge\ \{\eta.\ \exists\mu.\ \mu\ U_r\ \eta \in old\ q' \wedge suffix\ 1\ \xi \models_r \eta\} \subseteq old\ q')$

lemma *L4-5*: *create-graph* $\varphi \leq$ *SPEC* (*create-graph-rslt-all* ξ)
⟨*proof*⟩

2.4 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*. $\mu\ U_r\ \eta$∈*old q* ⟶ η∈*old q*}|μ η. $\mu\ U_r\ \eta \in$ *subfrmlsr* φ},
gba-L = λ*q l*. *q*∈*qs* ∧ {*p*. *prop*$_r$(*p*)∈*old q*}⊆*l* ∧ {*p*. *nprop*$_r$(*p*)∈*old q*} ∩ *l* = {}
⦈

end

locale *create-gba-from-nodes-precond* =
fixes φ :: *'a ltlr*
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*:
ipath gba.E r $\longleftrightarrow$ ($\forall$ *i. r i* $\in$*qs* $\wedge$ *name* (*r i*)$\in$*incoming* (*r* (*Suc i*)))
⟨*proof*⟩

lemma *create-gba-from-nodes--is-run*:
gba.is-run r $\longleftrightarrow$ *expand-init* $\in$ *incoming* (*r 0*)
$\wedge$ ($\forall$ *i. r i*$\in$*qs* $\wedge$ *name* (*r i*)$\in$*incoming* (*r* (*Suc i*)))
⟨*proof*⟩

context
begin

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

fun *auto-run* :: [*'a interprt*, *'a node set*] $\Rightarrow$ *'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*$_r$ $\notin$ *old* (*fst n-ns*) $\wedge$ ($\forall$ *nd*$\in$*snd n-ns. false*$_r$ $\notin$ *old nd*)
(**is** *?Q n-ns*)
shows *expand n-ns* $\leq$ *SPEC* (λ*nm-nds*. $\forall$ *nd*$\in$*snd nm-nds. false*$_r$ $\notin$ *old nd*)
⟨*proof*⟩

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

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

lemma *expand-or-propag*:
assumes μ *or*$_r$ $\eta \in$ *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*$_r$ $\eta \in$ *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*:
create-graph $\varphi \leq$ *SPEC* (λ*nds*. $\forall$ *nd*$\in$*nds*. μ *or*$_r$ $\eta \in$ *old nd* $\longrightarrow \{\mu, \eta\} \cap$ *old nd* $\neq \{\}$)
⟨*proof*⟩

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

abbreviation
next-propag--rslt μ *ns* $\equiv$
$\forall$ *nd*$\in$*ns*. $\forall$ *nd*$'\in$*ns*. X_r $\mu \in$ *old nd* $\wedge$ *name nd* $\in$ *incoming nd*$'$ $\longrightarrow$ $\mu \in$ *old nd*$'$

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

lemma *next-propag-on-create-graph*:
create-graph $\varphi \leq$ *SPEC* (λ*nds*. $\forall$ *n*$\in$*nds*. $\forall$ *n*$'\in$*nds*. X_r $\mu \in$*old n* $\wedge$ *name n*$\in$*incoming*

$n' \longrightarrow \mu \in old\ n')$
⟨*proof*⟩

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

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

lemma *expand-release-propag*:
fixes *n-ns* :: - × *'a node set*
assumes *release-propag--assm* $\mu\ \eta$ *n-ns*
$\wedge$ *release-propag--rslt* $\mu\ \eta$ (*snd n-ns*)
$\wedge$ *expand-assm-incoming n-ns*
$\wedge$ *expand-name-ident* (*snd n-ns*) (**is** *?Q n-ns*)
shows *expand n-ns* ≤ *SPEC* ($\lambda r.$ *release-propag--rslt* $\mu\ \eta$ (*snd r*))
(**is** - ≤ *SPEC ?P*)
⟨*proof*⟩

lemma *release-propag-on-create-graph*:
create-graph φ
≤ *SPEC* ($\lambda nds.\ \forall\, n \in nds.\ \forall\, n' \in nds.\ \mu\ R_r\ \eta \in old\ n \wedge name\ n \in incoming\ n'$
$\longrightarrow (\{\mu, \eta\} \subseteq old\ n \vee \eta \in old\ n \wedge \mu\ R_r\ \eta \in old\ n')$)
⟨*proof*⟩

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

abbreviation
until-propag--rslt f g ns ≡

$\forall\, n{\in}ns.\ \forall\, nd{\in}ns.\ f\ U_r\ g \in old\ n \wedge name\ n \in incoming\ nd$
$\longrightarrow (g \in old\ n \vee (f{\in}old\ n \wedge f\ U_r\ g \in old\ nd))$

lemma *expand-until-propag*:
fixes *n-ns* :: - × *'a node set*
assumes *until-propag--assm* $\mu\ \eta$ *n-ns*
$\wedge$ *until-propag--rslt* $\mu\ \eta$ (*snd n-ns*)
$\wedge$ *expand-assm-incoming n-ns*
$\wedge$ *expand-name-ident* (*snd n-ns*) (**is** *?Q n-ns*)
shows *expand n-ns* ≤ *SPEC* (λr. *until-propag--rslt* $\mu\ \eta$ (*snd r*))
(**is** - ≤ *SPEC ?P*)
⟨*proof*⟩

lemma *until-propag-on-create-graph*:
create-graph φ ≤ *SPEC* ($\lambda nds.\ \forall\, n{\in}nds.\ \forall\, n'{\in}nds.\ \mu\ U_r\ \eta{\in}old\ n \wedge name\ n{\in}incoming\ n'$
$\longrightarrow (\eta{\in}old\ n \vee \mu{\in}old\ n \wedge \mu\ U_r\ \eta{\in}old\ n'))$
⟨*proof*⟩

definition *all-subfrmls* :: *'a node* ⇒ *'a frml set*
where *all-subfrmls n* ≡ $\bigcup$(*subfrmlsr* ' (*new n* ∪ *old n* ∪ *next n*))

lemma *all-subfrmls--UnionD*:
assumes ($\bigcup x{\in}A.$ *subfrmlsr x*) ⊆ *B*
and *x*∈*A*
and *y*∈*subfrmlsr x*
shows *y*∈*B*
⟨*proof*⟩

lemma *expand-all-subfrmls-propag*:
assumes *all-subfrmls* (*fst n-ns*) ⊆ *B* ∧ ($\forall\, nd{\in}snd\ n\text{-}ns.$ *all-subfrmls nd* ⊆ *B*) (**is** *?Q n-ns*)
shows *expand n-ns* ≤ *SPEC* ($\lambda r.\ \forall\, nd{\in}snd\ r.$ *all-subfrmls nd* ⊆ *B*)
(**is** - ≤ *SPEC ?P*)
⟨*proof*⟩

lemma *old-propag-on-create-graph*: *create-graph* φ ≤ *SPEC* ($\lambda nds.\ \forall\, n{\in}nds.$ *old n* ⊆ *subfrmlsr* φ)
⟨*proof*⟩

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

lemma *L4-2a*:

assumes *ipath gba.E σ*
and $\mu\ U_r\ \eta \in$ *old* $(\sigma\ 0)$
shows $(\forall i.\ \{\mu, \mu\ U_r\ \eta\} \subseteq$ *old* $(\sigma\ i) \wedge \eta \notin$ *old* $(\sigma\ i))$
$\vee\ (\exists j.\ (\forall i{<}j.\ \{\mu, \mu\ U_r\ \eta\} \subseteq$ *old* $(\sigma\ i)) \wedge \eta \in$ *old* $(\sigma\ j))$
(**is** *?A* $\vee$ *?B*)
⟨*proof*⟩

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

lemma *L4-2c*:
assumes *run*: *ipath gba.E σ*
and $\mu\ R_r\ \eta \in$ *old* $(\sigma\ 0)$
shows $\forall i.\ \eta \in$ *old* $(\sigma\ i) \vee (\exists j{<}i.\ \mu \in$ *old* $(\sigma\ j))$
⟨*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* $(\sigma\ i)\ (\xi\ i)$ (**is** *?lgbarel-accept* $\xi\ \sigma$)
and $\psi \in$ *old* $(\sigma\ 0)$
shows $\xi \models_r \psi$
⟨*proof*⟩

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

lemma *expand-expand-init-propag*:
assumes $(\forall$ *nm*$\in$*incoming n′*. *nm* $<$ *name n′*$)$
$\wedge$ *name n′* $\leq$ *name* $($*fst n-ns*$)$
$\wedge$ $($*incoming n′* $\cap$ *incoming* $($*fst n-ns*$) \neq \{\}$
$\longrightarrow$ *new n′* $\subseteq$ *old* $($*fst n-ns*$)\ \cup$ *new* $($*fst n-ns*$))$
(**is** *?Q n-ns*)
and $\forall$ *nd*$\in$*snd n-ns*. $\forall$ *nm*$\in$*incoming n′*. *nm*$\in$*incoming nd* $\longrightarrow$ *new n′* $\subseteq$ *old nd*
(**is** *?P* $($*snd n-ns*$)$)
shows *expand n-ns* $\leq$ *SPEC* $(\lambda r.$ *name n′*$\leq$*fst r* $\wedge$ *?P* $($*snd r*$))$
⟨*proof*⟩

lemma *expand-init-propag-on-create-graph*:
create-graph $\varphi \leq$ *SPEC*$(\lambda$*nds*. $\forall$ *nd*$\in$*nds*. *expand-init*$\in$*incoming nd* $\longrightarrow \varphi \in$ *old*

nd)
⟨*proof*⟩

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

lemma *L4-9*:
assumes *acc*: *gba.accept ξ*
shows *ξ* ⊨$_r$ *φ*
⟨*proof*⟩

lemma *L4-10*:
assumes *ξ* ⊨$_r$ *φ*
shows *gba.accept ξ*
⟨*proof*⟩

end
end

lemma *create-graph--name-ident*: *create-graph φ* ≤ *SPEC* (λ*nds*. ∀ *q*∈*nds*. ∃!*q′*∈*nds*. *name q* = *name q′*)
⟨*proof*⟩

corollary *create-graph--name-inj*: *create-graph φ* ≤ *SPEC* (λ*nds*. *inj-on name nds*)
⟨*proof*⟩

definition
create-gba φ
≡ *do* { *nds* ← *create-graph*$_T$ *φ*;
RETURN (*create-gba-from-nodes φ nds*) }

lemma *create-graph-precond*: *create-graph φ*
≤ *SPEC* (*create-gba-from-nodes-precond φ*)
⟨*proof*⟩

lemma *create-gba--invar*: *create-gba φ* ≤ *SPEC gba*
⟨*proof*⟩

lemma *create-gba-acc*:
shows *create-gba φ* ≤ *SPEC*(λ*A*. ∀ *ξ*. *gba.accept A ξ* ⟷ *ξ* ⊨$_r$ *φ*)
⟨*proof*⟩

lemma *create-gba--name-inj*:
shows *create-gba φ* ≤ *SPEC*(λ*A*. (*inj-on name* (*g-V A*)))
⟨*proof*⟩

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

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

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

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

lemma *create-gba--no-init*:
shows *create-gba* $\varphi \leq SPEC(\lambda\mathcal{A}.\ expand\text{-}init \notin name\ `\ (g\text{-}V\ \mathcal{A}))$
⟨*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* $\varphi \leq SPEC\ (\lambda\mathcal{A}.\ nds\text{-}invars\ (g\text{-}V\ \mathcal{A}))$
⟨*proof*⟩

theorem *T4-1*:
create-gba $\varphi \leq SPEC($
$\lambda\mathcal{A}.\ gba\ \mathcal{A}$
$\wedge\ finite\ (g\text{-}V\ \mathcal{A})$
$\wedge\ (\forall\,\xi.\ gba.accept\ \mathcal{A}\ \xi \longleftrightarrow \xi \models_r \varphi)$
$\wedge\ (nds\text{-}invars\ (g\text{-}V\ \mathcal{A})))$
⟨*proof*⟩

definition *create-name-gba* $\varphi \equiv do\ \{$
$G \leftarrow$ *create-gba* φ;
ASSERT (*nds-invars* (*g-V G*));
RETURN (*gba-rename name G*)
}

theorem *create-name-gba-correct*:

create-name-gba $\varphi \leq$ *SPEC*($\lambda\mathcal{A}$. *gba* $\mathcal{A} \wedge$ *finite* (*g-V* $\mathcal{A}$) $\wedge$ ($\forall\xi$. *gba.accept* $\mathcal{A}$ ξ $\longleftrightarrow$ $\xi \models_r \varphi$))
⟨*proof*⟩

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

lemma *create-name-igba-correct*: *create-name-igba* $\varphi \leq$ *SPEC* (λ*G*.
igba G $\wedge$ *finite* (*g-V G*) $\wedge$ ($\forall\xi$. *igba.accept G* ξ $\longleftrightarrow$ $\xi \models_r \varphi$))
⟨*proof*⟩

context
notes [*refine-vcg*] = *order-trans*[*OF create-name-gba-correct*]
begin

lemma *create-name-igba* $\varphi \leq$ *SPEC* (λ*G*. *igba G* $\wedge$ ($\forall\xi$. *igba.accept G* ξ $\longleftrightarrow$ $\xi \models_r \varphi$))
⟨*proof*⟩

end

end

3 Refinement to Efficient Code

theory *LTL-to-GBA-impl*
imports
LTL-to-GBA
Deriving.Compare-Instances
CAVA-Automata.Automata-Impl
CAVA-Base.CAVA-Code-Target
begin

3.1 Parametricity Setup Boilerplate

3.1.1 LTL Formulas

derive *linorder ltlr*

inductive-set *ltlr-rel* **for** *R* **where**
(*True-ltlr*, *True-ltlr*) ∈ *ltlr-rel R*
| (*False-ltlr*, *False-ltlr*) ∈ *ltlr-rel R*
| (*a*,*a'*)∈*R* ⟹ (*Prop-ltlr a*,*Prop-ltlr a'*) ∈ *ltlr-rel R*

| $(a,a')\in R \Longrightarrow (Nprop\text{-}ltlr\ a, Nprop\text{-}ltlr\ a') \in ltlr\text{-}rel\ R$
| $[\![(a,a')\in ltlr\text{-}rel\ R;\ (b,b')\in ltlr\text{-}rel\ R]\!]$
$\Longrightarrow (And\text{-}ltlr\ a\ b, And\text{-}ltlr\ a'\ b') \in ltlr\text{-}rel\ R$
| $[\![(a,a')\in ltlr\text{-}rel\ R;\ (b,b')\in ltlr\text{-}rel\ R]\!]$
$\Longrightarrow (Or\text{-}ltlr\ a\ b, Or\text{-}ltlr\ a'\ b') \in ltlr\text{-}rel\ R$
| $[\![(a,a')\in ltlr\text{-}rel\ R]\!] \Longrightarrow (Next\text{-}ltlr\ a, Next\text{-}ltlr\ a') \in ltlr\text{-}rel\ R$
| $[\![(a,a')\in ltlr\text{-}rel\ R;\ (b,b')\in ltlr\text{-}rel\ R]\!]$
$\Longrightarrow (Until\text{-}ltlr\ a\ b, Until\text{-}ltlr\ a'\ b') \in ltlr\text{-}rel\ R$
| $[\![(a,a')\in ltlr\text{-}rel\ R;\ (b,b')\in ltlr\text{-}rel\ R]\!]$
$\Longrightarrow (Release\text{-}ltlr\ a\ b, Release\text{-}ltlr\ a'\ b') \in ltlr\text{-}rel\ R$

lemmas *ltlr-rel-induct*[*induct set*]
$=$ *ltlr-rel.induct*[*simplified relAPP-def*[*of ltlr-rel*, *symmetric*]]
lemmas *ltlr-rel-cases*[*cases set*]
$=$ *ltlr-rel.cases*[*simplified relAPP-def*[*of ltlr-rel*, *symmetric*]]
lemmas *ltlr-rel-intros*
$=$ *ltlr-rel.intros*[*simplified relAPP-def*[*of ltlr-rel*, *symmetric*]]

inductive-simps *ltlr-rel-left-simps*[*simplified relAPP-def*[*of ltlr-rel*, *symmetric*]]:
$(True\text{-}ltlr,z) \in ltlr\text{-}rel\ R$
$(False\text{-}ltlr,z) \in ltlr\text{-}rel\ R$
$(Prop\text{-}ltlr\ p,\ z) \in ltlr\text{-}rel\ R$
$(Nprop\text{-}ltlr\ p,\ z) \in ltlr\text{-}rel\ R$
$(And\text{-}ltlr\ a\ b,\ z) \in ltlr\text{-}rel\ R$
$(Or\text{-}ltlr\ a\ b,\ z) \in ltlr\text{-}rel\ R$
$(Next\text{-}ltlr\ a,\ z) \in ltlr\text{-}rel\ R$
$(Until\text{-}ltlr\ a\ b,\ z) \in ltlr\text{-}rel\ R$
$(Release\text{-}ltlr\ a\ b,\ z) \in ltlr\text{-}rel\ R$

lemma *ltlr-rel-sv*[*relator-props*]:
assumes *SV*: *single-valued R*
shows *single-valued* $(\langle R\rangle ltlr\text{-}rel)$
⟨*proof*⟩

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

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

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

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

lemma *ltlr-con-param*[*param*, *autoref-rules*]:
$(True\text{-}ltlr,\ True\text{-}ltlr) \in \langle R\rangle ltlr\text{-}rel$
$(False\text{-}ltlr,\ False\text{-}ltlr) \in \langle R\rangle ltlr\text{-}rel$
$(Prop\text{-}ltlr,\ Prop\text{-}ltlr) \in R \to \langle R\rangle ltlr\text{-}rel$
$(Nprop\text{-}ltlr,\ Nprop\text{-}ltlr) \in R \to \langle R\rangle ltlr\text{-}rel$

$(\mathit{And\text{-}ltlr},\ \mathit{And\text{-}ltlr}) \in \langle R\rangle \mathit{ltlr\text{-}rel} \to \langle R\rangle \mathit{ltlr\text{-}rel} \to \langle R\rangle \mathit{ltlr\text{-}rel}$
$(\mathit{Or\text{-}ltlr},\ \mathit{Or\text{-}ltlr}) \in \langle R\rangle \mathit{ltlr\text{-}rel} \to \langle R\rangle \mathit{ltlr\text{-}rel} \to \langle R\rangle \mathit{ltlr\text{-}rel}$
$(\mathit{Next\text{-}ltlr},\ \mathit{Next\text{-}ltlr}) \in \langle R\rangle \mathit{ltlr\text{-}rel} \to \langle R\rangle \mathit{ltlr\text{-}rel}$
$(\mathit{Until\text{-}ltlr},\ \mathit{Until\text{-}ltlr}) \in \langle R\rangle \mathit{ltlr\text{-}rel} \to \langle R\rangle \mathit{ltlr\text{-}rel} \to \langle R\rangle \mathit{ltlr\text{-}rel}$
$(\mathit{Release\text{-}ltlr},\ \mathit{Release\text{-}ltlr}) \in \langle R\rangle \mathit{ltlr\text{-}rel} \to \langle R\rangle \mathit{ltlr\text{-}rel} \to \langle R\rangle \mathit{ltlr\text{-}rel}$
⟨*proof*⟩

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

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

lemma *case-ltlr-mono*[*refine-mono*]:
assumes $\varphi = \mathit{True\text{-}ltlr} \Longrightarrow a \leq a'$
assumes $\varphi = \mathit{False\text{-}ltlr} \Longrightarrow b \leq b'$
assumes $\bigwedge p.\ \varphi = \mathit{Prop\text{-}ltlr}\ p \Longrightarrow c\ p \leq c'\ p$
assumes $\bigwedge p.\ \varphi = \mathit{Nprop\text{-}ltlr}\ p \Longrightarrow d\ p \leq d'\ p$
assumes $\bigwedge \mu\ \nu.\ \varphi = \mathit{And\text{-}ltlr}\ \mu\ \nu \Longrightarrow e\ \mu\ \nu \leq e'\ \mu\ \nu$
assumes $\bigwedge \mu\ \nu.\ \varphi = \mathit{Or\text{-}ltlr}\ \mu\ \nu \Longrightarrow f\ \mu\ \nu \leq f'\ \mu\ \nu$
assumes $\bigwedge \mu.\ \varphi = \mathit{Next\text{-}ltlr}\ \mu \Longrightarrow g\ \mu \leq g'\ \mu$
assumes $\bigwedge \mu\ \nu.\ \varphi = \mathit{Until\text{-}ltlr}\ \mu\ \nu \Longrightarrow h\ \mu\ \nu \leq h'\ \mu\ \nu$
assumes $\bigwedge \mu\ \nu.\ \varphi = \mathit{Release\text{-}ltlr}\ \mu\ \nu \Longrightarrow i\ \mu\ \nu \leq i'\ \mu\ \nu$
shows $\mathit{case\text{-}ltlr}\ a\ b\ c\ d\ e\ f\ g\ h\ i\ \varphi \leq \mathit{case\text{-}ltlr}\ a'\ b'\ c'\ d'\ e'\ f'\ g'\ h'\ i'\ \varphi$
⟨*proof*⟩

primrec *ltlr-eq* **where**
$\mathit{ltlr\text{-}eq}\ \mathit{eq}\ \mathit{True\text{-}ltlr}\ f \longleftrightarrow (\mathit{case}\ f\ \mathit{of}\ \mathit{True\text{-}ltlr} \Rightarrow \mathit{True} \mid \text{-} \Rightarrow \mathit{False})$
$\mid \mathit{ltlr\text{-}eq}\ \mathit{eq}\ \mathit{False\text{-}ltlr}\ f \longleftrightarrow (\mathit{case}\ f\ \mathit{of}\ \mathit{False\text{-}ltlr} \Rightarrow \mathit{True} \mid \text{-} \Rightarrow \mathit{False})$
$\mid \mathit{ltlr\text{-}eq}\ \mathit{eq}\ (\mathit{Prop\text{-}ltlr}\ p)\ f \longleftrightarrow (\mathit{case}\ f\ \mathit{of}\ \mathit{Prop\text{-}ltlr}\ p' \Rightarrow \mathit{eq}\ p\ p' \mid \text{-} \Rightarrow \mathit{False})$
$\mid \mathit{ltlr\text{-}eq}\ \mathit{eq}\ (\mathit{Nprop\text{-}ltlr}\ p)\ f \longleftrightarrow (\mathit{case}\ f\ \mathit{of}\ \mathit{Nprop\text{-}ltlr}\ p' \Rightarrow \mathit{eq}\ p\ p' \mid \text{-} \Rightarrow \mathit{False})$
$\mid \mathit{ltlr\text{-}eq}\ \mathit{eq}\ (\mathit{And\text{-}ltlr}\ \mu\ \nu)\ f$
$\longleftrightarrow (\mathit{case}\ f\ \mathit{of}\ \mathit{And\text{-}ltlr}\ \mu'\ \nu' \Rightarrow \mathit{ltlr\text{-}eq}\ \mathit{eq}\ \mu\ \mu' \wedge \mathit{ltlr\text{-}eq}\ \mathit{eq}\ \nu\ \nu' \mid \text{-} \Rightarrow \mathit{False})$

| *ltlr-eq eq* (*Or-ltlr* μ ν) *f*
  $\longleftrightarrow$ (*case f of Or-ltlr* μ' ν' $\Rightarrow$ *ltlr-eq eq* μ μ' $\wedge$ *ltlr-eq eq* ν ν' | - $\Rightarrow$ *False*)
| *ltlr-eq eq* (*Next-ltlr* φ) *f*
  $\longleftrightarrow$ (*case f of Next-ltlr* φ' $\Rightarrow$ *ltlr-eq eq* φ φ' | - $\Rightarrow$ *False*)
| *ltlr-eq eq* (*Until-ltlr* μ ν) *f*
  $\longleftrightarrow$ (*case f of Until-ltlr* μ' ν' $\Rightarrow$ *ltlr-eq eq* μ μ' $\wedge$ *ltlr-eq eq* ν ν' | - $\Rightarrow$ *False*)
| *ltlr-eq eq* (*Release-ltlr* μ ν) *f*
  $\longleftrightarrow$ (*case f of*
    Release-ltlr μ' ν' $\Rightarrow$ *ltlr-eq eq* μ μ' $\wedge$ *ltlr-eq eq* ν ν'
  | - $\Rightarrow$ *False*)

lemma *ltlr-eq-autoref*[*autoref-rules*]:
  assumes *EQP*: (*eq*,(=)) $\in$ *R* $\rightarrow$ *R* $\rightarrow$ *bool-rel*
  shows (*ltlr-eq eq*, (=)) $\in$ $\langle$*R*$\rangle$*ltlr-rel* $\rightarrow$ $\langle$*R*$\rangle$*ltlr-rel* $\rightarrow$ *bool-rel*
⟨*proof*⟩

lemma *ltlr-dflt-cmp*[*autoref-rules-raw*]:
  assumes *PREFER-id R*
  shows
  (*dflt-cmp* ($\leq$) ($<$), *dflt-cmp* ($\leq$) ($<$))
  $\in$ $\langle$*R*$\rangle$*ltlr-rel* $\rightarrow$ $\langle$*R*$\rangle$*ltlr-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-ltlr-gtransfer*:
  assumes
  γ *ai* $\leq$ *a*
  γ *bi* $\leq$ *b*
  $\bigwedge$*a*. γ (*ci a*) $\leq$ *c a*
  $\bigwedge$*a*. γ (*di a*) $\leq$ *d a*
  $\bigwedge$*ltlr1 ltlr2*. γ (*ei ltlr1 ltlr2*) $\leq$ *e ltlr1 ltlr2*
  $\bigwedge$*ltlr1 ltlr2*. γ (*fi ltlr1 ltlr2*) $\leq$ *f ltlr1 ltlr2*
  $\bigwedge$*ltlr*. γ (*gi ltlr*) $\leq$ *g ltlr*
  $\bigwedge$*ltlr1 ltlr2*. γ (*hi ltlr1 ltlr2*) $\leq$ *h ltlr1 ltlr2*
  $\bigwedge$*ltlr1 ltlr2*. γ (*iiv ltlr1 ltlr2*) $\leq$ *i ltlr1 ltlr2*
  shows γ (*case-ltlr ai bi ci di ei fi gi hi iiv* φ)
    $\leq$ (*case-ltlr a b c d e f g h i* φ)
  ⟨*proof*⟩

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

lemma [*refine-transfer*]:
  assumes
  ai $\neq$ *dSUCCEED*

bi ≠ *dSUCCEED*
⋀*a*. *ci a* ≠ *dSUCCEED*
⋀*a*. *di a* ≠ *dSUCCEED*
⋀*ltlr1 ltlr2*. *ei ltlr1 ltlr2* ≠ *dSUCCEED*
⋀*ltlr1 ltlr2*. *fi ltlr1 ltlr2* ≠ *dSUCCEED*
⋀*ltlr*. *gi ltlr* ≠ *dSUCCEED*
⋀*ltlr1 ltlr2*. *hi ltlr1 ltlr2* ≠ *dSUCCEED*
⋀*ltlr1 ltlr2*. *iiv ltlr1 ltlr2* ≠ *dSUCCEED*
shows *case-ltlr ai bi ci di ei fi gi hi iiv φ* ≠ *dSUCCEED*
⟨*proof*⟩

3.1.2 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*⟩*ltlr-rel*⟩*lss.rel*
∧ (*oldi*,*old*)∈⟨⟨*R*⟩*ltlr-rel*⟩*lss.rel*
∧ (*nexti*,*next*)∈⟨⟨*R*⟩*ltlr-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*,

$\ldots = morei$
$|)$,
$(|$ $name = name$,
$incoming = inc$,
$new{=}new$,
$old{=}old$,
$next = next$,
$\ldots = more$
$|)$) $|$ $namei\ name\ inci\ inc\ newi\ new\ oldi\ old\ nexti\ next\ morei\ more.$
$(namei,name) \in node\text{-}name\text{-}rel$
$\wedge\ (inci,inc) \in \langle node\text{-}name\text{-}rel \rangle dflt\text{-}rs\text{-}rel$
$\wedge\ (newi,new) \in \langle \langle R \rangle ltlr\text{-}rel \rangle lss.rel$
$\wedge\ (oldi,old) \in \langle \langle R \rangle ltlr\text{-}rel \rangle lss.rel$
$\wedge\ (nexti,next) \in \langle \langle R \rangle ltlr\text{-}rel \rangle lss.rel$
$\wedge\ (morei,more) \in Re$
$\}$ $\langle proof \rangle$

lemma *node-rel-sv*[*relator-props*]:
$single\text{-}valued\ Re \Longrightarrow single\text{-}valued\ R \Longrightarrow single\text{-}valued\ (\langle Re,R \rangle node\text{-}rel)$
$\langle proof \rangle$

consts *i-node* :: $interface \Rightarrow interface \Rightarrow interface$

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

lemma [*autoref-rules*]: $(node\text{-}impl\text{-}ext,\ node\text{-}ext) \in$
$node\text{-}name\text{-}rel$
$\rightarrow \langle node\text{-}name\text{-}rel \rangle dflt\text{-}rs\text{-}rel$
$\rightarrow \langle \langle R \rangle ltlr\text{-}rel \rangle lss.rel$
$\rightarrow \langle \langle R \rangle ltlr\text{-}rel \rangle lss.rel$
$\rightarrow \langle \langle R \rangle ltlr\text{-}rel \rangle lss.rel$
$\rightarrow Re$
$\rightarrow \langle Re,R \rangle node\text{-}rel$
$\langle proof \rangle$

lemma [*autoref-rules*]:
$(node\text{-}impl.name\text{-}impl\text{-}update, node.name\text{-}update)$
$\in (node\text{-}name\text{-}rel \rightarrow node\text{-}name\text{-}rel) \rightarrow \langle Re,R \rangle node\text{-}rel \rightarrow \langle Re,R \rangle node\text{-}rel$
$(node\text{-}impl.incoming\text{-}impl\text{-}update, node.incoming\text{-}update)$
$\in (\langle node\text{-}name\text{-}rel \rangle dflt\text{-}rs\text{-}rel \rightarrow \langle node\text{-}name\text{-}rel \rangle dflt\text{-}rs\text{-}rel)$
$\rightarrow \langle Re,R \rangle node\text{-}rel$
$\rightarrow \langle Re,R \rangle node\text{-}rel$
$(node\text{-}impl.new\text{-}impl\text{-}update, node.new\text{-}update)$
$\in (\langle \langle R \rangle ltlr\text{-}rel \rangle lss.rel \rightarrow \langle \langle R \rangle ltlr\text{-}rel \rangle lss.rel) \rightarrow \langle Re,R \rangle node\text{-}rel \rightarrow \langle Re,R \rangle node\text{-}rel$
$(node\text{-}impl.old\text{-}impl\text{-}update, node.old\text{-}update)$
$\in (\langle \langle R \rangle ltlr\text{-}rel \rangle lss.rel \rightarrow \langle \langle R \rangle ltlr\text{-}rel \rangle lss.rel) \rightarrow \langle Re,R \rangle node\text{-}rel \rightarrow \langle Re,R \rangle node\text{-}rel$
$(node\text{-}impl.next\text{-}impl\text{-}update, node.next\text{-}update)$
$\in (\langle \langle R \rangle ltlr\text{-}rel \rangle lss.rel \rightarrow \langle \langle R \rangle ltlr\text{-}rel \rangle lss.rel) \rightarrow \langle Re,R \rangle node\text{-}rel \rightarrow \langle Re,R \rangle node\text{-}rel$

```
  (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⟩ltlr-rel⟩lss.rel
  (node-impl.old-impl,node.old)∈⟨Re,R⟩node-rel → ⟨⟨R⟩ltlr-rel⟩lss.rel
  (node-impl.next-impl,node.next)∈⟨Re,R⟩node-rel → ⟨⟨R⟩ltlr-rel⟩lss.rel
  (node-impl.more, node.more)∈⟨Re,R⟩node-rel → Re
  ⟨proof⟩
```

3.2 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.

3.2.1 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)
  }



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_r(q) ⇒
          if nprop_r(q)∈old n then RETURN (name n, ns)
          else expand (n⦇ old := {φ} ∪ old n ⦈, ns)
      | nprop_r(q) ⇒
          if prop_r(q)∈old n then RETURN (name n, ns)
          else expand (n⦇ old := {φ} ∪ old n ⦈, ns)
      | true_r ⇒ expand (n⦇ old := {φ} ∪ old n ⦈, ns)
      | false_r ⇒ RETURN (name n, ns)
      | ν and_r μ ⇒ expand (n⦇
          new := insert ν (insert μ (new n)),
          old := {φ} ∪ old n,
          next := next n ⦈, ns)
      | X_r ν ⇒ expand
          (n⦇ new := new n, old := {φ} ∪ old n, next := insert ν (next n) ⦈, ns)
      | μ or_r ν ⇒ expand2 expand n ns φ μ {} {ν}
      | μ U_r ν ⇒ expand2 expand n ns φ μ {φ} {ν}
      | μ R_r 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)         }
    }
  )
```

lemma *expand-aimpl-refine*:
  fixes *n-ns* :: ('*a node* × -)
  defines $R \equiv Id \cap \{(\text{-},(n,ns)).\ \forall n'{\in}ns.\ n > name\ n'\}$
  defines $R' \equiv Id \cap \{(\text{-},(n,ns)).\ \forall n'{\in}ns.\ name\ n > name\ n'\}$
  assumes [*refine*]: $(n\text{-}ns', n\text{-}ns) \in R'$
  shows $expand\text{-}aimpl\ n\text{-}ns' \le \Downarrow R\ (expand_T\ n\text{-}ns)$
  ⟨*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* $\varphi \leq \Downarrow Id$ (*create-graph*$_T$ φ)
⟨*proof*⟩

3.2.2 Creation of GBA from Nodes

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

lemma *create-name-gba-alt*: *create-name-gba* φ = *do* {
nds ← *create-graph*$_T$ φ;
ASSERT (*nds-invars nds*);
RETURN (*gba-rename-ext* (λ-. ()) *name* (*create-gba-from-nodes* φ *nds*))
}
⟨*proof*⟩

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** ($\lambda qn.$ *dflt-None-set* (*?L qn*)) = ($\lambda qn.$ *dflt-None-set* (*?R qn*)))
⟨*proof*⟩

lemma *build-succ-correct*:
assumes *NINIT*: *expand-init* $\notin$ *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* ($\lambda r.$
E-of-succ ($\lambda qn.$ *the-default* {} (*r qn*))
= *rename-E* ($\lambda u.$ *name u*) {(*q*, *q'*). *q* $\in$ *nds* $\wedge$ *q'* $\in$ *nds* $\wedge$ *name q* $\in$ *incoming q'*})
⟨*proof*⟩

primrec *until-frmlsr* :: *'a frml* $\Rightarrow$ (*'a frml* $\times$ *'a frml*) *set* **where**
until-frmlsr (μ *and*$_r$ ψ) = (*until-frmlsr* μ) $\cup$ (*until-frmlsr* ψ)
| *until-frmlsr* (*X*$_r$ μ) = *until-frmlsr* μ
| *until-frmlsr* (μ *U*$_r$ ψ) = *insert* (μ, ψ) ((*until-frmlsr* μ) $\cup$ (*until-frmlsr* ψ))
| *until-frmlsr* (μ *R*$_r$ ψ) = (*until-frmlsr* μ) $\cup$ (*until-frmlsr* ψ)
| *until-frmlsr* (μ *or*$_r$ ψ) = (*until-frmlsr* μ) $\cup$ (*until-frmlsr* ψ)
| *until-frmlsr* (*true*$_r$) = {}
| *until-frmlsr* (*false*$_r$) = {}
| *until-frmlsr* (*prop*$_r$(-)) = {}
| *until-frmlsr* (*nprop*$_r$(-)) = {}

lemma *until-frmlsr-correct*:
until-frmlsr φ = {(μ, η). *Until-ltlr* μ η $\in$ *subfrmlsr* φ}
⟨*proof*⟩

definition *build-F nds* φ
$\equiv$ ($\lambda(\mu,\eta).$ *name* ' {*q* $\in$ *nds*. (*Until-ltlr* μ η $\in$ *old q* $\longrightarrow$ η $\in$ *old q*)}) '
until-frmlsr φ

lemma *build-F-correct*: *build-F nds* φ =
{*name* ' *A* |*A*. $\exists \mu$ η. *A* = {*q* $\in$ *nds*. *Until-ltlr* μ η $\in$ *old q* $\longrightarrow$ η $\in$ *old q*} $\wedge$
Until-ltlr μ η $\in$ *subfrmlsr* φ}
⟨*proof*⟩

definition *pn-props ps* $\equiv$ *FOREACHi*
(λit (*P*,*N*). *P* = {*p*. *Prop-ltlr p* $\in$ *ps* $-$ *it*} $\wedge$ *N* = {*p*. *Nprop-ltlr p* $\in$ *ps* $-$ *it*})
ps (λp (*P*,*N*).
case p of Prop-ltlr p $\Rightarrow$ *RETURN* (*insert p P*,*N*)
| *Nprop-ltlr p* $\Rightarrow$ *RETURN* (*P*, *insert p N*)
| - $\Rightarrow$ *RETURN* (*P*,*N*)

) ({},{})

lemma *pn-props-correct*:
assumes [*simp*]: *finite ps*
shows *pn-props ps* ≤ *SPEC*(λ*r*. *r* =
({*p*. *Prop-ltlr p* ∈ *ps*}, {*p*. *Nprop-ltlr p* ∈ *ps*}))
⟨*proof*⟩

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

lemma *pn-map-correct*:
assumes [*simp*]: *finite nds*
assumes *FIN'*: ⋀*nd*. *nd*∈*nds* ⟹ *finite* (*old nd*)
assumes *INJ*: *inj-on name nds*
shows *pn-map nds* ≤ *SPEC* (λ*r*. ∀ *qn*.
case r qn of
None ⇒ *qn* ∉ *name*‘*nds*
| *Some* (*P*,*N*) ⇒ *qn* ∈ *name*‘*nds*
∧ *P* = {*p*. *Prop-ltlr p* ∈ *old* (*the-inv-into nds name qn*)}
∧ *N* = {*p*. *Nprop-ltlr p* ∈ *old* (*the-inv-into nds name qn*)}
)
⟨*proof*⟩

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

lemma *cr-rename-gba-refine*:
assumes *INV*: *nds-invars nds*
assumes *REL*[*simplified*]: (*nds'*,*nds*)∈*Id* (φ',φ)∈*Id*
shows *cr-rename-gba nds'* φ'
≤ ⇓*Id* (*RETURN* (*gba-rename-ext* (λ-. ()) *name* (*create-gba-from-nodes* φ *nds*)))
⟨*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*⟩

3.3 Refinement to Efficient Data Structures

3.3.1 Creation of GBA from Nodes

schematic-goal *until-frmlsr-impl-aux*:
  assumes [*relator-props*, *simp*]: *R*=*Id*
  shows (*?c*,*until-frmlsr*)
  $\in \langle$(*R*::(-×-::*linorder*) *set*)$\rangle$*ltlr-rel* → $\langle\langle R\rangle$*ltlr-rel* ×$_r$ $\langle R\rangle$*ltlr-rel*$\rangle$*dflt-rs-rel*
  ⟨*proof*⟩
concrete-definition *until-frmlsr-impl* **uses** *until-frmlsr-impl-aux*
lemmas [*autoref-rules*] = *until-frmlsr-impl.refine*[*OF PREFER-id-D*]

schematic-goal *build-succ-impl-aux*:
  shows (*?c*,*build-succ*) ∈
    $\langle\langle Rm,R\rangle$*node-rel*$\rangle$*list-set-rel*
    → $\langle\langle$*nat-rel*,$\langle$*nat-rel*$\rangle$*list-set-rel*$\rangle$*iam-map-rel*$\rangle$*nres-rel*
  ⟨*proof*⟩

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

schematic-goal *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-goal *build-F-impl-aux*:
  assumes [*relator-props*]: *R* = *Id*
  shows (*?c*,*build-F*) ∈

$\langle\langle Rm,R\rangle node\text{-}rel\rangle list\text{-}set\text{-}rel \to \langle R\rangle ltlr\text{-}rel \to \langle\langle nat\text{-}rel\rangle list\text{-}set\text{-}rel\rangle list\text{-}set\text{-}rel$
⟨*proof*⟩

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

schematic-goal *pn-map-impl-aux*:
shows (*?c*,*pn-map*) ∈
$\langle\langle Rm,Id\rangle node\text{-}rel\rangle list\text{-}set\text{-}rel$
$\to \langle\langle nat\text{-}rel,\langle Id\rangle list\text{-}set\text{-}rel \times_r \langle Id\rangle list\text{-}set\text{-}rel\rangle iam\text{-}map\text{-}rel\rangle nres\text{-}rel$
⟨*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*) ∈
$\langle\langle Rm,R\rangle node\text{-}rel\rangle list\text{-}set\text{-}rel$
$\to \langle\langle nat\text{-}rel,\langle R\rangle list\text{-}set\text{-}rel \times_r \langle R\rangle list\text{-}set\text{-}rel\rangle iam\text{-}map\text{-}rel\rangle nres\text{-}rel$
⟨*proof*⟩

schematic-goal *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*

schematic-goal *cr-rename-gba-impl-aux*:
assumes *ID*[*relator-props*]: *R*=*Id*
notes [*autoref-tyrel del*] = *tyrel-dflt-linorder-set*
notes [*autoref-tyrel*] = *ty-REL*[*of* ⟨*nat-rel*⟩*list-set-rel*]
shows (*?c*,*cr-rename-gba*) ∈
$\langle\langle Rm,R\rangle node\text{-}rel\rangle list\text{-}set\text{-}rel \to \langle R\rangle ltlr\text{-}rel \to$ (*?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*) ∈
$\langle\langle Rm,\ R\rangle node\text{-}rel\rangle list\text{-}set\text{-}rel \to \langle R\rangle ltlr\text{-}rel \to$
$\langle gbav\text{-}impl\text{-}rel\text{-}ext\ unit\text{-}rel\ nat\text{-}rel\ (\langle R\rangle fun\text{-}set\text{-}rel)\rangle nres\text{-}rel$
⟨*proof*⟩

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

3.3.2 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*) ∈ *Ra* → *Rb*
assumes *SIDE-PRECOND* (∀ *x*. *name* (*f x*) = *name x*)
shows (*map fi*, (*OP* (‘) ::: (*Ra*→*Rb*) → ⟨*Ra*⟩*ls-nds-rel* → ⟨*Rb*⟩*ls-nds-rel*)$*f*)
∈ ⟨*Ra*⟩*ls-nds-rel* → ⟨*Rb*⟩*ls-nds-rel*
⟨*proof*⟩

lemma *list-set-autoref-to-list*[*autoref-ga-rules*]:
shows *is-set-to-sorted-list* (λ- -. *True*) *R ls-nds-rel id*
⟨*proof*⟩

end

context begin interpretation *autoref-syn* ⟨*proof*⟩
lemma [*autoref-itype*]:
upd-incoming
::$_i$ ⟨*Im*, *I*⟩$_i$*i-node* →$_i$ ⟨⟨*Im'*, *I*⟩$_i$*i-node*⟩$_i$*i-set* →$_i$ ⟨⟨*Im'*, *I*⟩$_i$*i-node*⟩$_i$*i-set*
⟨*proof*⟩
end

term *upd-incoming*
schematic-goal *upd-incoming-impl-aux*:
assumes *REL-IS-ID R*
shows (*?c*, *upd-incoming*)∈⟨*Rm1*,*R*⟩*node-rel*
→ ⟨⟨*Rm2*,*R*⟩*node-rel*⟩*ls-nds-rel*
→ ⟨⟨*Rm2*,*R*⟩*node-rel*⟩*ls-nds-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-goal *expand-impl-aux*: (*?c*, *expand-aimpl*) ∈
⟨*unit-rel*,*Id*⟩*node-rel* ×$_r$ ⟨⟨*unit-rel*,*Id*⟩*node-rel*⟩*ls-nds-rel*
→ ⟨*nat-rel* ×$_r$ ⟨⟨*unit-rel*,*Id*⟩*node-rel*⟩*ls-nds-rel*⟩*nres-rel*
⟨*proof*⟩

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

context begin interpretation *autoref-syn* ⟨*proof*⟩
lemma [*autoref-itype*]: *expand*$_T$
::$_i$ ⟨⟨*i-unit*, *I*⟩$_i$*i-node*, ⟨⟨*i-unit*, *I*⟩$_i$*i-node*⟩$_i$*i-set*⟩$_i$*i-prod*
→$_i$ ⟨⟨*i-nat*,⟨⟨*i-unit*, *I*⟩$_i$*i-node*⟩$_i$*i-set*⟩$_i$*i-prod*⟩$_i$*i-nres* ⟨*proof*⟩

lemma *expand-autoref*[*autoref-rules*]:
assumes *ID*: *PREFER-id R*
assumes *A*: (*n-ns'*, *n-ns*)
∈ ⟨*unit-rel*,*R*⟩*node-rel* ×$_r$ ⟨⟨*unit-rel*,*R*⟩*node-rel*⟩*list-set-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*

$\rightarrow \langle$*nat-rel* $\times_r$ $\langle\langle$*unit-rel*,*R*$\rangle$*node-rel*$\rangle$*list-set-rel*$\rangle$*nres-rel*$)$\$*n-ns*$)$
$\in \langle$*nat-rel* $\times_r$ $\langle\langle$*unit-rel*,*R*$\rangle$*node-rel*$\rangle$*list-set-rel*$\rangle$*nres-rel*
⟨*proof*⟩

end

schematic-goal *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-goal *create-graph-impl-aux*:
assumes *ID*: *R*=*Id*
shows (*?c*, *create-graph-aimpl*)
∈ ⟨*R*⟩*ltlr-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-goal *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-goal *create-name-gba-impl-aux*:
(*?c*, (*create-name-gba-aimpl*:: *'a*::*linorder ltlr* ⇒ -))
∈ ⟨*Id*⟩*ltlr-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*⟩*ltlr-rel* → ⟨*gbav-impl-rel-ext unit-rel nat-rel* (⟨*R*⟩*fun-set-rel*)⟩*nres-rel*
(**is** -:-→⟨*?R*⟩*nres-rel*)
⟨*proof*⟩

schematic-goal *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*

schematic-goal *create-name-igba-impl-aux*:
assumes *RID*: *R=Id*
shows (*?c*,*create-name-igba*)∈
⟨*R*⟩*ltlr-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-goal *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://isa-afp.org/entries/Stuttering_Equivalence.shtml, Formal proof development.