Transition Systems and Automata

Julian Brunner

May 26, 2024

Abstract

This entry provides a very abstract theory of transition systems that can be instantiated to express various types of automata. A transition system is typically instantiated by providing a set of initial states, a predicate for enabled transitions, and a transition execution function. From this, it defines the concepts of finite and infinite paths as well as the set of reachable states, among other things. Many useful theorems, from basic path manipulation rules to coinduction and run construction rules, are proven in this abstract transition system context. The library comes with instantiations for DFAs, NFAs, and Büchi automata.

Contents

1 Basics

theory *Basic*
imports *Main*
begin

1.1 Miscellaneous

abbreviation (*input*) *const* $x \equiv \lambda$ -. x

lemmas [*simp*] = *map-prod.id map-prod.comp*[*symmetric*]
lemma *prod-UNIV*[*iff*]: $A \times B = UNIV \longleftrightarrow A = UNIV \land B = UNIV$ ⟨*proof*⟩

lemma *prod-singleton*:
fst ' $A = \{x\} \Longrightarrow A = fst$ ' $A \times snd$ ' A
snd ' $A = \{y\} \Longrightarrow A = fst$ ' $A \times snd$ ' A
⟨*proof*⟩

lemma *infinite-subset*[*trans*]: *infinite* $A \Longrightarrow A \subseteq B \Longrightarrow$ *infinite* B ⟨*proof*⟩
lemma *finite-subset*[*trans*]: $A \subseteq B \Longrightarrow$ *finite* $B \Longrightarrow$ *finite* A ⟨*proof*⟩

declare *infinite-coinduct*[*case-names infinite*, *coinduct pred*: *infinite*]
lemma *infinite-psubset-coinduct*[*case-names infinite*, *consumes 1*]:
assumes $R\ A$
assumes $\bigwedge A.\ R\ A \Longrightarrow \exists\ B \subset A.\ R\ B$
shows *infinite* A
⟨*proof*⟩

thm *inj-on-subset subset-inj-on*

lemma *inj-inj-on*[*dest*]: *inj* $f \Longrightarrow$ *inj-on* $f\ S$ ⟨*proof*⟩

end

2 Finite and Infinite Sequences

theory *Sequence*
imports
Basic
HOL−Library.Stream
HOL−Library.Monad-Syntax
begin

2.1 List Basics

declare *upt-Suc*[*simp del*]
declare *last.simps*[*simp del*]
declare *butlast.simps*[*simp del*]
declare *Cons-nth-drop-Suc*[*simp*]
declare *list.pred-True*[*simp*]

lemma *list-pred-cases*:
assumes *list-all* $P\ xs$
obtains (*nil*) $xs = []$ | (*cons*) $y\ ys$ **where** $xs = y\ \#\ ys$ $P\ y$ *list-all* $P\ ys$
⟨*proof*⟩

lemma *lists-iff-set*: $w \in$ *lists* $A \longleftrightarrow$ *set* $w \subseteq A$ ⟨*proof*⟩

lemma *fold-const*: *fold const* $xs\ a$ = *last* $(a\ \#\ xs)$
⟨*proof*⟩

lemma *take-Suc*: *take* (*Suc n*) *xs* = (*if xs* = [] *then* [] *else hd xs* # *take n* (*tl xs*))
⟨*proof*⟩

lemma *bind-map*[*simp*]: *map f xs* ⪢= *g* = *xs* ⪢= *g* ∘ *f* ⟨*proof*⟩

lemma *ball-bind*[*iff*]: *Ball* (*set* (*xs* ⪢= *f*)) *P* ⟷ (∀ *x* ∈ *set xs*. ∀ *y* ∈ *set* (*f x*). *P y*)
⟨*proof*⟩
lemma *bex-bind*[*iff*]: *Bex* (*set* (*xs* ⪢= *f*)) *P* ⟷ (∃ *x* ∈ *set xs*. ∃ *y* ∈ *set* (*f x*). *P y*)
⟨*proof*⟩

lemma *list-choice*: *list-all* (λ *x*. ∃ *y*. *P x y*) *xs* ⟷ (∃ *ys*. *list-all2 P xs ys*)
⟨*proof*⟩

lemma *listset-member*: *ys* ∈ *listset XS* ⟷ *list-all2* (∈) *ys XS*
⟨*proof*⟩
lemma *listset-empty*[*iff*]: *listset XS* = {} ⟷ ¬ *list-all* (λ *A*. *A* ≠ {}) *XS*
⟨*proof*⟩
lemma *listset-finite*[*iff*]:
assumes *list-all* (λ *A*. *A* ≠ {}) *XS*
shows *finite* (*listset XS*) ⟷ *list-all finite XS*
⟨*proof*⟩
lemma *listset-finite′*[*intro*]:
assumes *list-all finite XS*
shows *finite* (*listset XS*)
⟨*proof*⟩
lemma *listset-card*[*simp*]: *card* (*listset XS*) = *prod-list* (*map card XS*)
⟨*proof*⟩

2.2 Stream Basics

declare *stream.map-id*[*simp*]
declare *stream.set-map*[*simp*]
declare *stream.set-sel*(*1*)[*intro*!, *simp*]
declare *stream.pred-True*[*simp*]
declare *stream.pred-map*[*iff*]
declare *stream.rel-map*[*iff*]
declare *shift-simps*[*simp del*]
declare *stake-sdrop*[*simp*]
declare *stake-siterate*[*simp del*]
declare *sdrop-snth*[*simp*]

lemma *stream-pred-cases*:
assumes *pred-stream P xs*
obtains (*scons*) *y ys* **where** *xs* = *y* ## *ys P y pred-stream P ys*
⟨*proof*⟩

lemma *stream-rel-coinduct*[*case-names stream-rel*, *coinduct pred*: *stream-all2*]:

assumes $R\ u\ v$
assumes $\bigwedge\ a\ u\ b\ v.\ R\ (a\ \#\#\ u)\ (b\ \#\#\ v) \Longrightarrow P\ a\ b \wedge R\ u\ v$
shows *stream-all2* $P\ u\ v$
⟨*proof*⟩
lemma *stream-rel-coinduct-shift*[*case-names stream-rel*, *consumes 1*]:
assumes $R\ u\ v$
assumes $\bigwedge\ u\ v.\ R\ u\ v \Longrightarrow$
$\exists\ u_1\ u_2\ v_1\ v_2.\ u = u_1\ @-\ u_2 \wedge v = v_1\ @-\ v_2 \wedge u_1 \neq [] \wedge v_1 \neq [] \wedge$ *list-all2* $P\ u_1\ v_1 \wedge R\ u_2\ v_2$
shows *stream-all2* $P\ u\ v$
⟨*proof*⟩

lemma *stream-pred-coinduct*[*case-names stream-pred*, *coinduct pred*: *pred-stream*]:
assumes $R\ w$
assumes $\bigwedge\ a\ w.\ R\ (a\ \#\#\ w) \Longrightarrow P\ a \wedge R\ w$
shows *pred-stream* $P\ w$
⟨*proof*⟩
lemma *stream-pred-coinduct-shift*[*case-names stream-pred*, *consumes 1*]:
assumes $R\ w$
assumes $\bigwedge\ w.\ R\ w \Longrightarrow \exists\ u\ v.\ w = u\ @-\ v \wedge u \neq [] \wedge$ *list-all* $P\ u \wedge R\ v$
shows *pred-stream* $P\ w$
⟨*proof*⟩
lemma *stream-pred-flat-coinduct*[*case-names stream-pred*, *consumes 1*]:
assumes $R\ ws$
assumes $\bigwedge\ w\ ws.\ R\ (w\ \#\#\ ws) \Longrightarrow w \neq [] \wedge$ *list-all* $P\ w \wedge R\ ws$
shows *pred-stream* P (*flat* ws)
⟨*proof*⟩

lemmas *stream-eq-coinduct*[*case-names stream-eq*, *coinduct pred*: *HOL.eq*] =
stream-rel-coinduct[**where** *?P* = *HOL.eq*, *unfolded stream.rel-eq*]
lemmas *stream-eq-coinduct-shift*[*case-names stream-eq*, *consumes 1*] =
stream-rel-coinduct-shift[**where** *?P* = *HOL.eq*, *unfolded stream.rel-eq list.rel-eq*]

lemma *stream-pred-shift*[*iff*]: *pred-stream* $P\ (u\ @-\ v) \longleftrightarrow$ *list-all* $P\ u \wedge$ *pred-stream* $P\ v$
⟨*proof*⟩
lemma *stream-rel-shift*[*iff*]:
assumes *length* u_1 = *length* v_1
shows *stream-all2* $P\ (u_1\ @-\ u_2)\ (v_1\ @-\ v_2) \longleftrightarrow$ *list-all2* $P\ u_1\ v_1 \wedge$ *stream-all2* $P\ u_2\ v_2$
⟨*proof*⟩

lemma *sset-subset-stream-pred*: *sset* $w \subseteq A \longleftrightarrow$ *pred-stream* $(\lambda\ a.\ a \in A)\ w$
⟨*proof*⟩

lemma *eq-scons*: $w = a\ \#\#\ v \longleftrightarrow a =$ *shd* $w \wedge v =$ *stl* w ⟨*proof*⟩
lemma *scons-eq*: $a\ \#\#\ v = w \longleftrightarrow$ *shd* $w = a \wedge$ *stl* $w = v$ ⟨*proof*⟩
lemma *eq-shift*: $w = u\ @-\ v \longleftrightarrow$ *stake* (*length* u) $w = u \wedge$ *sdrop* (*length* u) $w = v$

⟨*proof*⟩
lemma *shift-eq*: u @– $v = w \longleftrightarrow u = stake\ (length\ u)\ w \wedge v = sdrop\ (length\ u)\ w$
⟨*proof*⟩
lemma *scons-eq-shift*: a ## $w = u$ @– $v \longleftrightarrow ([] = u \wedge a$ ## $w = v) \vee (\exists\ u'.\ a$ # $u' = u \wedge w = u'$ @– $v)$
⟨*proof*⟩
lemma *shift-eq-scons*: u @– $v = a$ ## $w \longleftrightarrow (u = [] \wedge v = a$ ## $w) \vee (\exists\ u'.\ u = a$ # $u' \wedge u'$ @– $v = w)$
⟨*proof*⟩

lemma *stream-all2-sset1*:
assumes *stream-all2 P xs ys*
shows $\forall\ x \in sset\ xs.\ \exists\ y \in sset\ ys.\ P\ x\ y$
⟨*proof*⟩
lemma *stream-all2-sset2*:
assumes *stream-all2 P xs ys*
shows $\forall\ y \in sset\ ys.\ \exists\ x \in sset\ xs.\ P\ x\ y$
⟨*proof*⟩

lemma *smap-eq-scons*[*iff*]: $smap\ f\ xs = y$ ## $ys \longleftrightarrow f\ (shd\ xs) = y \wedge smap\ f\ (stl\ xs) = ys$
⟨*proof*⟩
lemma *scons-eq-smap*[*iff*]: y ## $ys = smap\ f\ xs \longleftrightarrow y = f\ (shd\ xs) \wedge ys = smap\ f\ (stl\ xs)$
⟨*proof*⟩
lemma *smap-eq-shift*[*iff*]:
$smap\ f\ w = u$ @– $v \longleftrightarrow (\exists\ w_1\ w_2.\ w = w_1$ @– $w_2 \wedge map\ f\ w_1 = u \wedge smap\ f\ w_2 = v)$
⟨*proof*⟩
lemma *shift-eq-smap*[*iff*]:
u @– $v = smap\ f\ w \longleftrightarrow (\exists\ w_1\ w_2.\ w = w_1$ @– $w_2 \wedge u = map\ f\ w_1 \wedge v = smap\ f\ w_2)$
⟨*proof*⟩

lemma *szip-eq-scons*[*iff*]: $szip\ xs\ ys = z$ ## $zs \longleftrightarrow (shd\ xs,\ shd\ ys) = z \wedge szip\ (stl\ xs)\ (stl\ ys) = zs$
⟨*proof*⟩
lemma *scons-eq-szip*[*iff*]: z ## $zs = szip\ xs\ ys \longleftrightarrow z = (shd\ xs,\ shd\ ys) \wedge zs = szip\ (stl\ xs)\ (stl\ ys)$
⟨*proof*⟩

lemma *siterate-eq-scons*[*iff*]: $siterate\ f\ s = a$ ## $w \longleftrightarrow s = a \wedge siterate\ f\ (f\ s) = w$
⟨*proof*⟩
lemma *scons-eq-siterate*[*iff*]: a ## $w = siterate\ f\ s \longleftrightarrow a = s \wedge w = siterate\ f\ (f\ s)$
⟨*proof*⟩

lemma *snth-0*: $(a \ \#\# \ w) \ !! \ 0 = a$ ⟨*proof*⟩
lemma *eqI-snth*:
  assumes $\bigwedge i. \ u \ !! \ i = v \ !! \ i$
  shows $u = v$
  ⟨*proof*⟩

lemma *stream-pred-snth*: *pred-stream* $P \ w \longleftrightarrow (\forall \ i. \ P \ (w \ !! \ i))$
  ⟨*proof*⟩
lemma *stream-rel-snth*: *stream-all2* $P \ u \ v \longleftrightarrow (\forall \ i. \ P \ (u \ !! \ i) \ (v \ !! \ i))$
⟨*proof*⟩

lemma *stream-rel-pred-szip*: *stream-all2* $P \ u \ v \longleftrightarrow$ *pred-stream* (*case-prod* P) (*szip* $u \ v$)
  ⟨*proof*⟩

lemma *sconst-eq*[*iff*]: *sconst* $x =$ *sconst* $y \longleftrightarrow x = y$ ⟨*proof*⟩
lemma *stream-pred--sconst*[*iff*]: *pred-stream* P (*sconst* x) $\longleftrightarrow P \ x$
  ⟨*proof*⟩
lemma *stream-rel-sconst*[*iff*]: *stream-all2* P (*sconst* x) (*sconst* y) $\longleftrightarrow P \ x \ y$
  ⟨*proof*⟩

lemma *set-sset-stake*[*intro*!, *simp*]: *set* (*stake* n *xs*) $\subseteq$ *sset xs*
  ⟨*proof*⟩
lemma *sset-sdrop*[*intro*!, *simp*]: *sset* (*sdrop* n *xs*) $\subseteq$ *sset xs*
  ⟨*proof*⟩

lemma *set-stake-snth*: $x \in$ *set* (*stake* n *xs*) $\longleftrightarrow (\exists \ i < n.$ *xs* $!! \ i = x)$
  ⟨*proof*⟩

lemma *szip-transfer*[*transfer-rule*]:
  includes *lifting-syntax*
  shows (*stream-all2* A ===> *stream-all2* B ===> *stream-all2* (*rel-prod* A B)) *szip szip*
  ⟨*proof*⟩
lemma *siterate-transfer*[*transfer-rule*]:
  includes *lifting-syntax*
  shows ((A ===> A) ===> A ===> *stream-all2* A) *siterate siterate*
  ⟨*proof*⟩

lemma *split-stream-first*:
  assumes $A \cap$ *sset xs* $\neq \{\}$
  obtains *ys a zs*
  where *xs* = *ys* @− a ## *zs* $A \cap$ *set ys* $= \{\}$ $a \in A$
⟨*proof*⟩
lemma *split-stream-first'*:
  assumes $x \in$ *sset xs*
  obtains *ys zs*

where $xs = ys$ @− x ## zs $x \notin set\ ys$
⟨*proof*⟩

lemma *streams-UNIV*[*iff*]: *streams* A = *UNIV* $\longleftrightarrow$ A = *UNIV*
⟨*proof*⟩
lemma *streams-int*[*simp*]: *streams* $(A \cap B)$ = *streams* $A \cap$ *streams* B ⟨*proof*⟩
lemma *streams-Int*[*simp*]: *streams* $(\bigcap\ S) = \bigcap$ (*streams* ‘ S) ⟨*proof*⟩

lemma *pred-list-listsp*[*pred-set-conv*]: *list-all* = *listsp*
⟨*proof*⟩
lemma *pred-stream-streamsp*[*pred-set-conv*]: *pred-stream* = *streamsp*
⟨*proof*⟩

2.3 The scan Function

primrec (*transfer*) *scan* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'b \Rightarrow 'b\ list$ **where**
scan f [] *a* = [] | *scan f* (*x* # *xs*) *a* = *f x a* # *scan f xs* (*f x a*)

lemma *scan-append*[*simp*]: *scan f* (*xs* @ *ys*) *a* = *scan f xs a* @ *scan f ys* (*fold f xs a*)
⟨*proof*⟩

lemma *scan-eq-nil*[*iff*]: *scan f xs a* = [] $\longleftrightarrow$ *xs* = [] ⟨*proof*⟩
lemma *scan-eq-cons*[*iff*]:
scan f xs a = *b* # *w* $\longleftrightarrow$ ($\exists$ *y ys*. *xs* = *y* # *ys* $\wedge$ *f y a* = *b* $\wedge$ *scan f ys* (*f y a*) = *w*)
⟨*proof*⟩
lemma *scan-eq-append*[*iff*]:
scan f xs a = *u* @ *v* $\longleftrightarrow$ ($\exists$ *ys zs*. *xs* = *ys* @ *zs* $\wedge$ *scan f ys a* = *u* $\wedge$ *scan f zs* (*fold f ys a*) = *v*)
⟨*proof*⟩

lemma *scan-length*[*simp*]: *length* (*scan f xs a*) = *length xs*
⟨*proof*⟩

lemma *scan-last*: *last* (*a* # *scan f xs a*) = *fold f xs a*
⟨*proof*⟩
lemma *scan-butlast*[*simp*]: *scan f* (*butlast xs*) *a* = *butlast* (*scan f xs a*)
⟨*proof*⟩

lemma *scan-const*[*simp*]: *scan const xs a* = *xs*
⟨*proof*⟩
lemma *scan-nth*[*simp*]:
assumes $i <$ *length* (*scan f xs a*)
shows *scan f xs a* ! *i* = *fold f* (*take* (*Suc i*) *xs*) *a*
⟨*proof*⟩
lemma *scan-map*[*simp*]: *scan f* (*map g xs*) *a* = *scan* ($f \circ g$) *xs a*
⟨*proof*⟩
lemma *scan-take*[*simp*]: *take k* (*scan f xs a*) = *scan f* (*take k xs*) *a*

⟨*proof*⟩

lemma *scan-drop*[*simp*]: $drop\ k\ (scan\ f\ xs\ a) = scan\ f\ (drop\ k\ xs)\ (fold\ f\ (take\ k\ xs)\ a)$

⟨*proof*⟩

primcorec (*transfer*) *sscan* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a\ stream \Rightarrow 'b \Rightarrow 'b\ stream$ **where**

$sscan\ f\ xs\ a = f\ (shd\ xs)\ a\ \#\#\ sscan\ f\ (stl\ xs)\ (f\ (shd\ xs)\ a)$

lemma *sscan-scons*[*simp*]: $sscan\ f\ (x\ \#\#\ xs)\ a = f\ x\ a\ \#\#\ sscan\ f\ xs\ (f\ x\ a)$

⟨*proof*⟩

lemma *sscan-shift*[*simp*]: $sscan\ f\ (xs\ @-\ ys)\ a = scan\ f\ xs\ a\ @-\ sscan\ f\ ys\ (fold\ f\ xs\ a)$

⟨*proof*⟩

lemma *sscan-eq-scons*[*iff*]:

$sscan\ f\ xs\ a = b\ \#\#\ w \longleftrightarrow f\ (shd\ xs)\ a = b \wedge sscan\ f\ (stl\ xs)\ (f\ (shd\ xs)\ a) = w$

⟨*proof*⟩

lemma *scons-eq-sscan*[*iff*]:

$b\ \#\#\ w = sscan\ f\ xs\ a \longleftrightarrow b = f\ (shd\ xs)\ a \wedge w = sscan\ f\ (stl\ xs)\ (f\ (shd\ xs)\ a)$

⟨*proof*⟩

lemma *sscan-const*[*simp*]: $sscan\ const\ xs\ a = xs$

⟨*proof*⟩

lemma *sscan-snth*[*simp*]: $sscan\ f\ xs\ a\ !!\ i = fold\ f\ (stake\ (Suc\ i)\ xs)\ a$

⟨*proof*⟩

lemma *sscan-scons-snth*[*simp*]: $(a\ \#\#\ sscan\ f\ xs\ a)\ !!\ i = fold\ f\ (stake\ i\ xs)\ a$

⟨*proof*⟩

lemma *sscan-smap*[*simp*]: $sscan\ f\ (smap\ g\ xs)\ a = sscan\ (f \circ g)\ xs\ a$

⟨*proof*⟩

lemma *sscan-stake*[*simp*]: $stake\ k\ (sscan\ f\ xs\ a) = scan\ f\ (stake\ k\ xs)\ a$

⟨*proof*⟩

lemma *sscan-sdrop*[*simp*]: $sdrop\ k\ (sscan\ f\ xs\ a) = sscan\ f\ (sdrop\ k\ xs)\ (fold\ f\ (stake\ k\ xs)\ a)$

⟨*proof*⟩

2.4 Transposing Streams

primcorec (*transfer*) *stranspose* :: $'a\ stream\ list \Rightarrow 'a\ list\ stream$ **where**

$stranspose\ ws = map\ shd\ ws\ \#\#\ stranspose\ (map\ stl\ ws)$

lemma *stranspose-eq-scons*[*iff*]: $stranspose\ ws = a\ \#\#\ w \longleftrightarrow map\ shd\ ws = a \wedge stranspose\ (map\ stl\ ws) = w$

⟨*proof*⟩

lemma *scons-eq-stranspose*[*iff*]: $a\ \#\#\ w = stranspose\ ws \longleftrightarrow a = map\ shd\ ws \wedge w = stranspose\ (map\ stl\ ws)$

⟨*proof*⟩

lemma *stranspose-nil*[*simp*]: *stranspose* [] = *sconst* [] ⟨*proof*⟩
lemma *stranspose-cons*[*simp*]: *stranspose* (*w* # *ws*) = *smap2 Cons w* (*stranspose ws*)
⟨*proof*⟩

lemma *snth-stranspose*[*simp*]: *stranspose ws* !! *k* = *map* (λ *w*. *w* !! *k*) *ws* ⟨*proof*⟩
lemma *stranspose-nth*[*simp*]:
assumes $k <$ *length ws*
shows *smap* (λ *xs*. *xs* ! *k*) (*stranspose ws*) = *ws* ! *k*
⟨*proof*⟩

2.5 Distinct Streams

coinductive *sdistinct* :: *'a stream* ⇒ *bool* **where**
scons[*intro*!]: $x \notin$ *sset xs* ⟹ *sdistinct xs* ⟹ *sdistinct* (*x* ## *xs*)

lemma *sdistinct-scons-elim*[*elim*!]:
assumes *sdistinct* (*x* ## *xs*)
obtains $x \notin$ *sset xs sdistinct xs*
⟨*proof*⟩

lemma *sdistinct-coinduct*[*case-names sdistinct*, *coinduct pred*: *sdistinct*]:
assumes *P xs*
assumes ⋀ *x xs*. *P* (*x* ## *xs*) ⟹ $x \notin$ *sset xs* ∧ *P xs*
shows *sdistinct xs*
⟨*proof*⟩

lemma *sdistinct-shift*[*intro*!]:
assumes *distinct xs sdistinct ys set xs* ∩ *sset ys* = {}
shows *sdistinct* (*xs* @− *ys*)
⟨*proof*⟩
lemma *sdistinct-shift-elim*[*elim*!]:
assumes *sdistinct* (*xs* @− *ys*)
obtains *distinct xs sdistinct ys set xs* ∩ *sset ys* = {}
⟨*proof*⟩

lemma *sdistinct-infinite-sset*:
assumes *sdistinct w*
shows *infinite* (*sset w*)
⟨*proof*⟩

lemma *not-sdistinct-decomp*:
assumes ¬ *sdistinct w*
obtains *u v a w'*
where *w* = *u* @− *a* ## *v* @− *a* ## *w'*
⟨*proof*⟩

2.6 Sorted Streams

coinductive (**in** *order*) *sascending* :: $'a$ *stream* $\Rightarrow$ *bool* **where**
$a \leq b \Longrightarrow$ *sascending* $(b \ \#\# \ w) \Longrightarrow$ *sascending* $(a \ \#\# \ b \ \#\# \ w)$

coinductive (**in** *order*) *sdescending* :: $'a$ *stream* $\Rightarrow$ *bool* **where**
$a \geq b \Longrightarrow$ *sdescending* $(b \ \#\# \ w) \Longrightarrow$ *sdescending* $(a \ \#\# \ b \ \#\# \ w)$

lemma *sdescending-coinduct*[*case-names sdescending*, *coinduct pred*: *sdescending*]:
assumes $P\ w$
assumes $\bigwedge a\ b\ w.\ P\ (a \ \#\# \ b \ \#\# \ w) \Longrightarrow a \geq b \wedge P\ (b \ \#\# \ w)$
shows *sdescending w*
⟨*proof*⟩

lemma *sdescending-scons*:
assumes *sdescending* $(a \ \#\# \ w)$
shows *sdescending w*
⟨*proof*⟩
lemma *sdescending-sappend*:
assumes *sdescending* $(u \ @- \ v)$
obtains *sdescending v*
⟨*proof*⟩
lemma *sdescending-sdrop*:
assumes *sdescending w*
shows *sdescending* (*sdrop k w*)
⟨*proof*⟩

lemma *sdescending-sset-scons*:
assumes *sdescending* $(a \ \#\# \ w)$
assumes $b \in$ *sset w*
shows $a \geq b$
⟨*proof*⟩
lemma *sdescending-sset-sappend*:
assumes *sdescending* $(u \ @- \ v)$
assumes $a \in$ *set u* $b \in$ *sset v*
shows $a \geq b$
⟨*proof*⟩

lemma *sdescending-snth-antimono*:
assumes *sdescending w*
shows *antimono* (*snth w*)
⟨*proof*⟩

lemma *sdescending-stuck*:
fixes w :: $'a$ :: *wellorder stream*
assumes *sdescending w*
obtains *u a*
where $w = u \ @- $ *sconst a*
⟨*proof*⟩

end

3 Linear Temporal Logic on Streams

theory *Sequence-LTL*
imports
Sequence
HOL−Library.Linear-Temporal-Logic-on-Streams
begin

3.1 Basics

Avoid destroying the constant *holds* prematurely.

lemmas [*simp del*] = *holds.simps holds-eq1 holds-eq2 not-holds-eq*

lemma *ev-smap*[*iff*]: *ev P* (*smap f w*) $\longleftrightarrow$ *ev* (*P* $\circ$ *smap f*) *w* ⟨*proof*⟩
lemma *alw-smap*[*iff*]: *alw P* (*smap f w*) $\longleftrightarrow$ *alw* (*P* $\circ$ *smap f*) *w* ⟨*proof*⟩
lemma *holds-smap*[*iff*]: *holds P* (*smap f w*) $\longleftrightarrow$ *holds* (*P* $\circ$ *f*) *w* ⟨*proof*⟩

lemmas [*iff*] = *ev-sconst alw-sconst hld-smap′*

lemmas [*iff*] = *alw-ev-stl*
lemma *alw-ev-sdrop*[*iff*]: *alw* (*ev P*) (*sdrop n w*) $\longleftrightarrow$ *alw* (*ev P*) *w*
⟨*proof*⟩
lemma *alw-ev-scons*[*iff*]: *alw* (*ev P*) (*a ## w*) $\longleftrightarrow$ *alw* (*ev P*) *w* ⟨*proof*⟩
lemma *alw-ev-shift*[*iff*]: *alw* (*ev P*) (*u @− v*) $\longleftrightarrow$ *alw* (*ev P*) *v* ⟨*proof*⟩

lemmas [*simp del*, *iff*] = *ev-alw-stl*
lemma *ev-alw-sdrop*[*iff*]: *ev* (*alw P*) (*sdrop n w*) $\longleftrightarrow$ *ev* (*alw P*) *w*
⟨*proof*⟩
lemma *ev-alw-scons*[*iff*]: *ev* (*alw P*) (*a ## w*) $\longleftrightarrow$ *ev* (*alw P*) *w* ⟨*proof*⟩
lemma *ev-alw-shift*[*iff*]: *ev* (*alw P*) (*u @− v*) $\longleftrightarrow$ *ev* (*alw P*) *v* ⟨*proof*⟩

lemma *holds-sconst*[*iff*]: *holds P* (*sconst a*) $\longleftrightarrow$ *P a* ⟨*proof*⟩
lemma *HLD-sconst*[*iff*]: *HLD A* (*sconst a*) $\longleftrightarrow$ *a* $\in$ *A* ⟨*proof*⟩

lemma *ev-alt-def*: *ev* φ *w* $\longleftrightarrow$ ($\exists$ *u v*. *w* = *u @− v* $\wedge$ φ *v*)
⟨*proof*⟩
lemma *ev-stl-alt-def*: *ev* φ (*stl w*) $\longleftrightarrow$ ($\exists$ *u v*. *w* = *u @− v* $\wedge$ *u* $\neq$ [] $\wedge$ φ *v*)
⟨*proof*⟩

lemma *ev-HLD-sset*: *ev* (*HLD A*) *w* $\longleftrightarrow$ *sset w* $\cap$ *A* $\neq$ {} ⟨*proof*⟩

lemma *alw-ev-coinduct*[*case-names alw-ev*, *consumes 1*]:
assumes *R w*
assumes $\bigwedge$ *w*. *R w* $\Longrightarrow$ *ev* φ *w* $\wedge$ *ev R* (*stl w*)

shows *alw* (*ev* φ) *w*
⟨*proof*⟩

3.2 Infinite Occurrence

abbreviation *infs P w* ≡ *alw* (*ev* (*holds P*)) *w*
abbreviation *fins P w* ≡ ¬ *infs P w*

lemma *infs-suffix*: *infs P w* ⟷ (∀ *u v*. *w* = *u* @− *v* ⟶ *Bex* (*sset v*) *P*)
⟨*proof*⟩
lemma *infs-snth*: *infs P w* ⟷ (∀ *n*. ∃ *k* ≥ *n*. *P* (*w* !! *k*))
⟨*proof*⟩
lemma *infs-infm*: *infs P w* ⟷ ($\exists_\infty$ *i*. *P* (*w* !! *i*))
⟨*proof*⟩

lemma *infs-coinduct*[*case-names infs*, *coinduct pred*: *infs*]:
assumes *R w*
assumes ⋀ *w*. *R w* ⟹ *Bex* (*sset w*) *P* ∧ *ev R* (*stl w*)
shows *infs P w*
⟨*proof*⟩
lemma *infs-coinduct-shift*[*case-names infs*, *consumes 1*]:
assumes *R w*
assumes ⋀ *w*. *R w* ⟹ ∃ *u v*. *w* = *u* @− *v* ∧ *Bex* (*set u*) *P* ∧ *R v*
shows *infs P w*
⟨*proof*⟩
lemma *infs-flat-coinduct*[*case-names infs-flat*, *consumes 1*]:
assumes *R w*
assumes ⋀ *u v*. *R* (*u* ## *v*) ⟹ *Bex* (*set u*) *P* ∧ *R v*
shows *infs P* (*flat w*)
⟨*proof*⟩
lemma *infs-sscan-coinduct*[*case-names infs-sscan*, *consumes 1*]:
assumes *R w a*
assumes ⋀ *w a*. *R w a* ⟹ *P a* ∧ (∃ *u v*. *w* = *u* @− *v* ∧ *u* ≠ [] ∧ *R v* (*fold f u a*))
shows *infs P* (*a* ## *sscan f w a*)
⟨*proof*⟩

lemma *infs-mono*: (⋀ *a*. *a* ∈ *sset w* ⟹ *P a* ⟹ *Q a*) ⟹ *infs P w* ⟹ *infs Q w*
⟨*proof*⟩
lemma *infs-mono-strong*: *stream-all2* (λ *a b*. *P a* ⟶ *Q b*) *u v* ⟹ *infs P u* ⟹ *infs Q v*
⟨*proof*⟩

lemma *infs-all*: *Ball* (*sset w*) *P* ⟹ *infs P w* ⟨*proof*⟩
lemma *infs-any*: *infs P w* ⟹ *Bex* (*sset w*) *P* ⟨*proof*⟩

lemma *infs-bot*[*iff*]: *infs bot w* ⟷ *False* ⟨*proof*⟩
lemma *infs-top*[*iff*]: *infs top w* ⟷ *True* ⟨*proof*⟩

lemma *infs-disj*[*iff*]: *infs* (λ *a*. *P a* $\vee$ *Q a*) *w* $\longleftrightarrow$ *infs P w* $\vee$ *infs Q w*
⟨*proof*⟩
lemma *infs-bex*[*iff*]:
assumes *finite S*
shows *infs* (λ *a*. $\exists$ *x* $\in$ *S*. *P x a*) *w* $\longleftrightarrow$ ($\exists$ *x* $\in$ *S*. *infs* (*P x*) *w*)
⟨*proof*⟩
lemma *infs-bex-le-nat*[*iff*]: *infs* (λ *a*. $\exists$ *k* < *n* :: *nat*. *P k a*) *w* $\longleftrightarrow$ ($\exists$ *k* < *n*. *infs* (*P k*) *w*)
⟨*proof*⟩

lemma *infs-cycle*[*iff*]:
assumes *w* $\neq$ []
shows *infs P* (*cycle w*) $\longleftrightarrow$ *Bex* (*set w*) *P*
⟨*proof*⟩

end

4 Zipping Sequences

theory *Sequence-Zip*
imports *Sequence-LTL*
begin

4.1 Zipping Lists

notation *zip* (**infixr** || *51*)

lemmas [*simp*] = *zip-map-fst-snd*

lemma *split-zip*[*no-atp*]: ($\bigwedge$ *x*. *PROP P x*) $\equiv$ ($\bigwedge$ *y z*. *length y* = *length z* $\Longrightarrow$ *PROP P* (*y* || *z*))
⟨*proof*⟩
lemma *split-zip-all*[*no-atp*]: ($\forall$ *x*. *P x*) $\longleftrightarrow$ ($\forall$ *y z*. *length y* = *length z* $\longrightarrow$ *P* (*y* || *z*))
⟨*proof*⟩
lemma *split-zip-ex*[*no-atp*]: ($\exists$ *x*. *P x*) $\longleftrightarrow$ ($\exists$ *y z*. *length y* = *length z* $\wedge$ *P* (*y* || *z*))
⟨*proof*⟩

lemma *zip-eq*[*iff*]:
assumes *length u* = *length v length r* = *length s*
shows *u* || *v* = *r* || *s* $\longleftrightarrow$ *u* = *r* $\wedge$ *v* = *s*
⟨*proof*⟩

lemma *list-rel-pred-zip*: *list-all2 P xs ys* $\longleftrightarrow$ *length xs* = *length ys* $\wedge$ *list-all* (*case-prod P*) (*xs* || *ys*)
⟨*proof*⟩

lemma *list-choice-zip*: *list-all* (λ *x*. $\exists$ *y*. *P x y*) *xs* $\longleftrightarrow$

$(\exists\ ys.\ length\ ys = length\ xs \wedge list\text{-}all\ (case\text{-}prod\ P)\ (xs\ \|\ ys))$
$\langle proof \rangle$
lemma *list-choice-pair*: $list\text{-}all\ (\lambda\ xy.\ case\text{-}prod\ (\lambda\ x\ y.\ \exists\ z.\ P\ x\ y\ z)\ xy)\ (xs\ \|\ ys) \longleftrightarrow$
$(\exists\ zs.\ length\ zs = min\ (length\ xs)\ (length\ ys) \wedge list\text{-}all\ (\lambda\ (x,\ y,\ z).\ P\ x\ y\ z)\ (xs\ \|\ ys\ \|\ zs))$
$\langle proof \rangle$

lemma *list-rel-zip*[*iff*]:
assumes $length\ u = length\ v\ length\ r = length\ s$
shows $list\text{-}all2\ (rel\text{-}prod\ A\ B)\ (u\ \|\ v)\ (r\ \|\ s) \longleftrightarrow list\text{-}all2\ A\ u\ r \wedge list\text{-}all2\ B\ v\ s$
$\langle proof \rangle$

lemma *zip-last*[*simp*]:
assumes $xs\ \|\ ys \neq []\ length\ xs = length\ ys$
shows $last\ (xs\ \|\ ys) = (last\ xs,\ last\ ys)$
$\langle proof \rangle$

4.2 Zipping Streams

notation *szip* (**infixr** ||| *51*)

lemmas [*simp*] = *szip-unfold*

lemma *smap-szip-same*: $smap\ f\ (xs\ |||\ xs) = smap\ (\lambda\ x.\ f\ (x,\ x))\ xs$ $\langle proof \rangle$

lemma *szip-smap*[*simp*]: $smap\ fst\ zs\ |||\ smap\ snd\ zs = zs$ $\langle proof \rangle$
lemma *szip-smap-fst*[*simp*]: $smap\ fst\ (xs\ |||\ ys) = xs$ $\langle proof \rangle$
lemma *szip-smap-snd*[*simp*]: $smap\ snd\ (xs\ |||\ ys) = ys$ $\langle proof \rangle$

lemma *szip-smap-both*: $smap\ f\ xs\ |||\ smap\ g\ ys = smap\ (map\text{-}prod\ f\ g)\ (xs\ |||\ ys)$
$\langle proof \rangle$
lemma *szip-smap-left*: $smap\ f\ xs\ |||\ ys = smap\ (apfst\ f)\ (xs\ |||\ ys)$ $\langle proof \rangle$
lemma *szip-smap-right*: $xs\ |||\ smap\ f\ ys = smap\ (apsnd\ f)\ (xs\ |||\ ys)$ $\langle proof \rangle$
lemmas *szip-smap-fold* = *szip-smap-both szip-smap-left szip-smap-right*

lemma *szip-sconst-smap-fst*: $sconst\ a\ |||\ xs = smap\ (Pair\ a)\ xs$
$\langle proof \rangle$
lemma *szip-sconst-smap-snd*: $xs\ |||\ sconst\ a = smap\ (prod.swap \circ Pair\ a)\ xs$
$\langle proof \rangle$

lemma *split-szip*[*no-atp*]: $(\bigwedge\ x.\ PROP\ P\ x) \equiv (\bigwedge\ y\ z.\ PROP\ P\ (y\ |||\ z))$
$\langle proof \rangle$
lemma *split-szip-all*[*no-atp*]: $(\forall\ x.\ P\ x) \longleftrightarrow (\forall\ y\ z.\ P\ (y\ |||\ z))$ $\langle proof \rangle$
lemma *split-szip-ex*[*no-atp*]: $(\exists\ x.\ P\ x) \longleftrightarrow (\exists\ y\ z.\ P\ (y\ |||\ z))$ $\langle proof \rangle$

lemma *szip-eq*[*iff*]: $u\ |||\ v = r\ |||\ s \longleftrightarrow u = r \wedge v = s$
$\langle proof \rangle$

lemma *stream-rel-szip*[*iff*]:
stream-all2 (*rel-prod A B*) (u ||| v) (r ||| s) $\longleftrightarrow$ *stream-all2 A u r* $\wedge$ *stream-all2 B v s*
⟨*proof*⟩

lemma *szip-shift*[*simp*]:
assumes *length* u = *length* s
shows u @− v ||| s @− t = (u || s) @− (v ||| t)
⟨*proof*⟩

lemma *szip-sset-fst*[*simp*]: *fst* ‘ *sset* (u ||| v) = *sset* u ⟨*proof*⟩
lemma *szip-sset-snd*[*simp*]: *snd* ‘ *sset* (u ||| v) = *sset* v ⟨*proof*⟩
lemma *szip-sset-elim*[*elim*]:
assumes (a, b) $\in$ *sset* (u ||| v)
obtains $a \in$ *sset* u $b \in$ *sset* v
⟨*proof*⟩
lemma *szip-sset*[*simp*]: *sset* (u ||| v) $\subseteq$ *sset* u $\times$ *sset* v ⟨*proof*⟩

lemma *sset-szip-finite*[*iff*]: *finite* (*sset* (u ||| v)) $\longleftrightarrow$ *finite* (*sset* u) $\wedge$ *finite* (*sset* v)
⟨*proof*⟩

lemma *infs-szip-fst*[*iff*]: *infs* ($P \circ$ *fst*) (u ||| v) $\longleftrightarrow$ *infs* P u
⟨*proof*⟩
lemma *infs-szip-snd*[*iff*]: *infs* ($P \circ$ *snd*) (u ||| v) $\longleftrightarrow$ *infs* P v
⟨*proof*⟩

end

5 Maps

theory *Maps*
imports *Sequence-Zip*
begin

6 Basics

lemma *fun-upd-None*[*simp*]:
assumes $p \notin$ *dom* f
shows f (p := *None*) = f
⟨*proof*⟩

lemma *finite-set-of-finite-maps'*:
assumes *finite* A *finite* B
shows *finite* $\{m.$ *dom* $m \subseteq A \wedge$ *ran* $m \subseteq B\}$
⟨*proof*⟩

lemma *fold-map-of*:
  assumes *distinct xs*
  shows $fold\ (\lambda\ x\ (k,\ m).\ (Suc\ k,\ m\ (x \mapsto k)))\ xs\ (k,\ m) =$
    $(k + length\ xs,\ m\ {+}{+}\ map\text{-}of\ (xs\ ||\ [k\ ..{<}\ k + length\ xs]))$
$\langle proof \rangle$

6.1 Expanding set functions to sets of functions

definition $expand :: ('a \Rightarrow 'b\ set) \Rightarrow ('a \Rightarrow 'b)\ set$ **where**
  $expand\ f = \{g.\ \forall\ x.\ g\ x \in f\ x\}$

lemma *expand-update*[*simp*]:
  assumes $f\ x \neq \{\}$
  shows $expand\ (f\ (x := S)) = (\bigcup\ y \in S.\ (\lambda\ g.\ g\ (x := y))\ `\ expand\ f)$
$\langle proof \rangle$

6.2 Expanding set maps into sets of maps

definition $expand\text{-}map :: ('a \rightharpoonup 'b\ set) \Rightarrow ('a \rightharpoonup 'b)\ set$ **where**
  $expand\text{-}map\ f \equiv expand\ (case\text{-}option\ \{None\}\ (image\ Some) \circ f)$

lemma *expand-map-alt-def*: $expand\text{-}map\ f =$
  $\{g.\ dom\ g = dom\ f \wedge (\forall\ x\ S\ y.\ f\ x = Some\ S \longrightarrow g\ x = Some\ y \longrightarrow y \in S)\}$
  $\langle proof \rangle$

lemma *expand-map-dom*:
  assumes $g \in expand\text{-}map\ f$
  shows $dom\ g = dom\ f$
  $\langle proof \rangle$

lemma *expand-map-empty*[*simp*]: $expand\text{-}map\ Map.empty = \{Map.empty\}$ $\langle proof \rangle$
lemma *expand-map-update*[*simp*]:
  $expand\text{-}map\ (f\ (x \mapsto S)) = (\bigcup\ y \in S.\ (\lambda\ g.\ g\ (x \mapsto y))\ `\ expand\text{-}map\ (f\ (x :=$
$None)))$
$\langle proof \rangle$

end
theory *Acceptance*
imports *Sequence-LTL*
begin

type-synonym $'a\ pred = 'a \Rightarrow bool$
type-synonym $'a\ rabin = 'a\ pred \times 'a\ pred$
type-synonym $'a\ gen = 'a\ list$

definition $rabin :: 'a\ rabin \Rightarrow 'a\ stream\ pred$ **where**
  $rabin \equiv \lambda\ (I,\ F)\ w.\ infs\ I\ w \wedge fins\ F\ w$

lemma *rabin*[*intro*]:

assumes $IF = (I,\ F)$ *infs I w fins F w*
shows *rabin IF w*
⟨*proof*⟩
lemma *rabin-elim*[*elim*]:
assumes *rabin IF w*
obtains *I F*
where $IF = (I,\ F)$ *infs I w fins F w*
⟨*proof*⟩

definition *gen* :: $('a \Rightarrow 'b\ pred) \Rightarrow ('a\ gen \Rightarrow 'b\ pred)$ **where**
$gen\ P\ cs\ w \equiv \forall\ \ c \in set\ cs.\ P\ c\ w$

lemma *gen*[*intro*]:
assumes $\bigwedge\ c.\ c \in set\ cs \Longrightarrow P\ c\ w$
shows *gen P cs w*
⟨*proof*⟩
lemma *gen-elim*[*elim*]:
assumes *gen P cs w*
obtains $\bigwedge\ c.\ c \in set\ cs \Longrightarrow P\ c\ w$
⟨*proof*⟩

definition *cogen* :: $('a \Rightarrow 'b\ pred) \Rightarrow ('a\ gen \Rightarrow 'b\ pred)$ **where**
$cogen\ P\ cs\ w \equiv \exists\ \ c \in set\ cs.\ P\ c\ w$

lemma *cogen*[*intro*]:
assumes $c \in set\ cs\ P\ c\ w$
shows *cogen P cs w*
⟨*proof*⟩
lemma *cogen-elim*[*elim*]:
assumes *cogen P cs w*
obtains *c*
where $c \in set\ cs\ P\ c\ w$
⟨*proof*⟩

lemma *cogen-alt-def*: $cogen\ P\ cs\ w \longleftrightarrow \neg\ gen\ (\lambda\ c\ w.\ Not\ (P\ c\ w))\ cs\ w$ ⟨*proof*⟩

end
theory *Degeneralization*
imports
Acceptance
Sequence-Zip
begin

type-synonym $'a\ degen = 'a \times nat$

definition *degen* :: $'a\ pred\ gen \Rightarrow 'a\ degen\ pred$ **where**
$degen\ cs \equiv \lambda\ (a,\ k).\ k \geq length\ cs \lor (cs\ !\ k)\ a$

lemma *degen-simps*[*iff*]: $degen\ cs\ (a,\ k) \longleftrightarrow k \geq length\ cs \lor (cs\ !\ k)\ a$ ⟨*proof*⟩

definition *count* :: *'a pred gen* ⇒ *'a* ⇒ *nat* ⇒ *nat* **where**
  count cs a k ≡
    if k < *length cs*
    then if (*cs* ! *k*) *a then Suc k mod length cs else k*
    else if cs = [] *then k else 0*

lemma *count-empty*[*simp*]: *count* [] *a k* = *k* ⟨*proof*⟩
lemma *count-nonempty*[*simp*]: *cs* ≠ [] ⟹ *count cs a k* < *length cs* ⟨*proof*⟩
lemma *count-constant-1*:
  assumes *k* < *length cs*
  assumes ⋀ *a*. *a* ∈ *set w* ⟹ ¬ (*cs* ! *k*) *a*
  shows *fold* (*count cs*) *w k* = *k*
  ⟨*proof*⟩
lemma *count-constant-2*:
  assumes *k* < *length cs*
  assumes ⋀ *a*. *a* ∈ *set* (*w* || *k* # *scan* (*count cs*) *w k*) ⟹ ¬ *degen cs a*
  shows *fold* (*count cs*) *w k* = *k*
  ⟨*proof*⟩
lemma *count-step*:
  assumes *k* < *length cs*
  assumes (*cs* ! *k*) *a*
  shows *count cs a k* = *Suc k mod length cs*
  ⟨*proof*⟩

lemma *degen-skip-condition*:
  assumes *k* < *length cs*
  assumes *infs* (*degen cs*) (*w* ||| *k* ## *sscan* (*count cs*) *w k*)
  obtains *u a v*
  where *w* = *u* @− *a* ## *v fold* (*count cs*) *u k* = *k* (*cs* ! *k*) *a*
⟨*proof*⟩
lemma *degen-skip-arbitrary*:
  assumes *k* < *length cs l* < *length cs*
  assumes *infs* (*degen cs*) (*w* ||| *k* ## *sscan* (*count cs*) *w k*)
  obtains *u v*
  where *w* = *u* @− *v fold* (*count cs*) *u k* = *l*
⟨*proof*⟩
lemma *degen-skip-arbitrary-condition*:
  assumes *l* < *length cs*
  assumes *infs* (*degen cs*) (*w* ||| *k* ## *sscan* (*count cs*) *w k*)
  obtains *u a v*
  where *w* = *u* @− *a* ## *v fold* (*count cs*) *u k* = *l* (*cs* ! *l*) *a*
⟨*proof*⟩
lemma *gen-degen-step*:
  assumes *gen infs cs w*
  obtains *u a v*
  where *w* = *u* @− *a* ## *v degen cs* (*a*, *fold* (*count cs*) *u k*)
⟨*proof*⟩

lemma *degen-infs*[*iff*]: *infs* (*degen cs*) (*w* ||| *k* ## *sscan* (*count cs*) *w k*) ⟷ *gen infs cs w*
⟨*proof*⟩

end

7 Transition Systems

theory *Transition-System*
imports ../*Basic*/*Sequence*
begin

7.1 Universal Transition Systems

locale *transition-system-universal* =
fixes *execute* :: *'transition* ⇒ *'state* ⇒ *'state*
begin

abbreviation *target* ≡ *fold execute*
abbreviation *states* ≡ *scan execute*
abbreviation *trace* ≡ *sscan execute*

lemma *target-alt-def*: *target r p* = *last* (*p* # *states r p*) ⟨*proof*⟩

end

7.2 Transition Systems

locale *transition-system* =
transition-system-universal execute
for *execute* :: *'transition* ⇒ *'state* ⇒ *'state*
\+
fixes *enabled* :: *'transition* ⇒ *'state* ⇒ *bool*
begin

abbreviation *successors p* ≡ {*execute a p* |*a*. *enabled a p*}

inductive *path* :: *'transition list* ⇒ *'state* ⇒ *bool* **where**
nil[*intro*!]: *path* [] *p* |
cons[*intro*!]: *enabled a p* ⟹ *path r* (*execute a p*) ⟹ *path* (*a* # *r*) *p*

inductive-cases *path-cons-elim*[*elim*!]: *path* (*a* # *r*) *p*

lemma *path-append*[*intro*!]:
assumes *path r p path s* (*target r p*)
shows *path* (*r* @ *s*) *p*
⟨*proof*⟩
lemma *path-append-elim*[*elim*!]:
assumes *path* (*r* @ *s*) *p*

obtains *path r p path s* (*target r p*)
⟨*proof*⟩

coinductive *run* :: $'$*transition stream* ⇒ $'$*state* ⇒ *bool* **where**
scons[*intro*!]: *enabled a p* ⟹ *run r* (*execute a p*) ⟹ *run* (*a* ## *r*) *p*

inductive-cases *run-scons-elim*[*elim*!]: *run* (*a* ## *r*) *p*

lemma *run-shift*[*intro*!]:
assumes *path r p run s* (*target r p*)
shows *run* (*r* @− *s*) *p*
⟨*proof*⟩
lemma *run-shift-elim*[*elim*!]:
assumes *run* (*r* @− *s*) *p*
obtains *path r p run s* (*target r p*)
⟨*proof*⟩

lemma *run-coinduct*[*case-names run*, *coinduct pred*: *run*]:
assumes *R r p*
assumes ⋀ *a r p*. *R* (*a* ## *r*) *p* ⟹ *enabled a p* ∧ *R r* (*execute a p*)
shows *run r p*
⟨*proof*⟩
lemma *run-coinduct-shift*[*case-names run*, *consumes 1*]:
assumes *R r p*
assumes ⋀ *r p*. *R r p* ⟹ ∃ *s t*. *r* = *s* @− *t* ∧ *s* ≠ [] ∧ *path s p* ∧ *R t* (*target s p*)
shows *run r p*
⟨*proof*⟩
lemma *run-flat-coinduct*[*case-names run*, *consumes 1*]:
assumes *R rs p*
assumes ⋀ *r rs p*. *R* (*r* ## *rs*) *p* ⟹ *r* ≠ [] ∧ *path r p* ∧ *R rs* (*target r p*)
shows *run* (*flat rs*) *p*
⟨*proof*⟩

inductive-set *reachable* :: $'$*state* ⇒ $'$*state set* **for** *p* **where**
reflexive[*intro*!]: *p* ∈ *reachable p* |
execute[*intro*!]: *q* ∈ *reachable p* ⟹ *enabled a q* ⟹ *execute a q* ∈ *reachable p*

inductive-cases *reachable-elim*[*elim*]: *q* ∈ *reachable p*

lemma *reachable-execute*$'$[*intro*]:
assumes *enabled a p q* ∈ *reachable* (*execute a p*)
shows *q* ∈ *reachable p*
⟨*proof*⟩
lemma *reachable-elim*$'$[*elim*]:
assumes *q* ∈ *reachable p*
obtains *q* = *p* | *a* **where** *enabled a p q* ∈ *reachable* (*execute a p*)
⟨*proof*⟩

lemma *reachable-target*[*intro*]:
assumes $q \in$ *reachable p path r q*
shows *target r q* $\in$ *reachable p*
⟨*proof*⟩
lemma *reachable-target-elim*[*elim*]:
assumes $q \in$ *reachable p*
obtains *r*
where *path r p q* = *target r p*
⟨*proof*⟩

lemma *reachable-alt-def*: *reachable p* = {*target r p* |*r*. *path r p*} ⟨*proof*⟩

lemma *reachable-trans*[*trans*]: $q \in$ *reachable p* $\Longrightarrow$ $s \in$ *reachable q* $\Longrightarrow$ $s \in$ *reachable p* ⟨*proof*⟩

lemma *reachable-successors*[*intro*!]: *successors p* $\subseteq$ *reachable p* ⟨*proof*⟩

lemma *reachable-step*: *reachable p* = *insert p* ($\bigcup$ (*reachable* ‘ *successors p*)) ⟨*proof*⟩

end

7.3 Transition Systems with Initial States

locale *transition-system-initial* =
transition-system execute enabled
for *execute* :: ′*transition* ⇒ ′*state* ⇒ ′*state*
and *enabled* :: ′*transition* ⇒ ′*state* ⇒ *bool*
+
fixes *initial* :: ′*state* ⇒ *bool*
begin

inductive-set *nodes* :: ′*state set* **where**
initial[*intro*]: *initial p* $\Longrightarrow$ $p \in$ *nodes* |
execute[*intro*!]: $p \in$ *nodes* $\Longrightarrow$ *enabled a p* $\Longrightarrow$ *execute a p* $\in$ *nodes*

lemma *nodes-target*[*intro*]:
assumes $p \in$ *nodes path r p*
shows *target r p* $\in$ *nodes*
⟨*proof*⟩
lemma *nodes-target-elim*[*elim*]:
assumes $q \in$ *nodes*
obtains *r p*
where *initial p path r p q* = *target r p*
⟨*proof*⟩

lemma *nodes-alt-def*: *nodes* = $\bigcup$ (*reachable* ‘ *Collect initial*) ⟨*proof*⟩

lemma *nodes-trans*[*trans*]: $p \in$ *nodes* $\Longrightarrow$ $q \in$ *reachable p* $\Longrightarrow$ $q \in$ *nodes* ⟨*proof*⟩

end

end

8 Additional Theorems for Transition Systems

theory *Transition-System-Extra*
imports
../Basic/Sequence-LTL
Transition-System
begin

context *transition-system*
begin

definition *enableds* $p \equiv \{a.\ enabled\ a\ p\}$
definition *paths* $p \equiv \{r.\ path\ r\ p\}$
definition *runs* $p \equiv \{r.\ run\ r\ p\}$

lemma *stake-run*:
assumes $\bigwedge k.\ path\ (stake\ k\ r)\ p$
shows *run r p*
⟨*proof*⟩
lemma *snth-run*:
assumes $\bigwedge k.\ enabled\ (r\ !!\ k)\ (target\ (stake\ k\ r)\ p)$
shows *run r p*
⟨*proof*⟩

lemma *run-stake*:
assumes *run r p*
shows $path\ (stake\ k\ r)\ p$
⟨*proof*⟩
lemma *run-sdrop*:
assumes *run r p*
shows $run\ (sdrop\ k\ r)\ (target\ (stake\ k\ r)\ p)$
⟨*proof*⟩
lemma *run-snth*:
assumes *run r p*
shows $enabled\ (r\ !!\ k)\ (target\ (stake\ k\ r)\ p)$
⟨*proof*⟩

lemma *run-alt-def-snth*: $run\ r\ p \longleftrightarrow (\forall\ k.\ enabled\ (r\ !!\ k)\ (target\ (stake\ k\ r)\ p))$
⟨*proof*⟩

lemma *reachable-states*:
assumes $q \in reachable\ p\ path\ r\ q$
shows $set\ (states\ r\ q) \subseteq reachable\ p$

⟨*proof*⟩
lemma *reachable-trace*:
assumes $q \in$ *reachable p run r q*
shows *sset* (*trace r q*) $\subseteq$ *reachable p*
⟨*proof*⟩

end

context *transition-system-initial*
begin

lemma *nodes-states*:
assumes $p \in$ *nodes path r p*
shows *set* (*states r p*) $\subseteq$ *nodes*
⟨*proof*⟩
lemma *nodes-trace*:
assumes $p \in$ *nodes run r p*
shows *sset* (*trace r p*) $\subseteq$ *nodes*
⟨*proof*⟩

end

end

9 Constructing Paths and Runs in Transition Systems

theory *Transition-System-Construction*
imports
../Basic/Sequence-LTL
Transition-System
begin

context *transition-system*
begin

lemma *invariant-run*:
assumes $P\ p$ $\bigwedge p.\ P\ p \Longrightarrow \exists\ a.\ enabled\ a\ p \wedge P\ (execute\ a\ p) \wedge Q\ p\ a$
obtains *r*
where *run r p pred-stream P* (*p* ## *trace r p*) *stream-all2 Q* (*p* ## *trace r p*) *r*
⟨*proof*⟩
lemma *recurring-condition*:
assumes $P\ p$ $\bigwedge p.\ P\ p \Longrightarrow \exists\ r.\ r \neq [] \wedge path\ r\ p \wedge P\ (target\ r\ p)$
obtains *r*
where *run r p infs P* (*p* ## *trace r p*)
⟨*proof*⟩

lemma *invariant-run-index*:
assumes $P\ n\ p\ \bigwedge n\ p.\ P\ n\ p \Longrightarrow \exists\ a.\ enabled\ a\ p \wedge P\ (Suc\ n)\ (execute\ a\ p) \wedge Q\ n\ p\ a$
obtains *r*
where
$run\ r\ p$
$\bigwedge i.\ P\ (n + i)\ (target\ (stake\ i\ r)\ p)$
$\bigwedge i.\ Q\ (n + i)\ (target\ (stake\ i\ r)\ p)\ (r\ !!\ i)$
⟨*proof*⟩

lemma *koenig*:
assumes $infinite\ (reachable\ p)$
assumes $\bigwedge q.\ q \in reachable\ p \Longrightarrow finite\ (successors\ q)$
obtains *r*
where $run\ r\ p$
⟨*proof*⟩

end

end

10 Deterministic Automata

theory *Deterministic*
imports
../*Transition-Systems*/*Transition-System*
../*Transition-Systems*/*Transition-System-Extra*
../*Transition-Systems*/*Transition-System-Construction*
../*Basic*/*Degeneralization*
begin

locale *automaton* =
fixes $automaton :: 'label\ set \Rightarrow 'state \Rightarrow ('label \Rightarrow 'state \Rightarrow 'state) \Rightarrow 'condition \Rightarrow 'automaton$
fixes *alphabet initial transition condition*
assumes *automaton*[*simp*]: $automaton\ (alphabet\ A)\ (initial\ A)\ (transition\ A)\ (condition\ A) = A$
assumes *alphabet*[*simp*]: $alphabet\ (automaton\ a\ i\ t\ c) = a$
assumes *initial*[*simp*]: $initial\ (automaton\ a\ i\ t\ c) = i$
assumes *transition*[*simp*]: $transition\ (automaton\ a\ i\ t\ c) = t$
assumes *condition*[*simp*]: $condition\ (automaton\ a\ i\ t\ c) = c$
begin

sublocale *transition-system-initial*
$transition\ A\ \lambda\ a\ p.\ a \in alphabet\ A\ \lambda\ p.\ p = initial\ A$
for *A*
defines $path' = path$ **and** $run' = run$ **and** $reachable' = reachable$ **and** $nodes' = nodes$

⟨*proof*⟩

lemma *path-alt-def*: *path A w p* ⟷ *w* ∈ *lists* (*alphabet A*)
⟨*proof*⟩
lemma *run-alt-def*: *run A w p* ⟷ *w* ∈ *streams* (*alphabet A*)
⟨*proof*⟩

end

locale *automaton-path* =
automaton automaton alphabet initial transition condition
for *automaton* :: *'label set* ⇒ *'state* ⇒ (*'label* ⇒ *'state* ⇒ *'state*) ⇒ *'condition* ⇒ *'automaton*
and *alphabet initial transition condition*
\+
fixes *test* :: *'condition* ⇒ *'label list* ⇒ *'state list* ⇒ *'state* ⇒ *bool*
begin

definition *language* :: *'automaton* ⇒ *'label list set* **where**
language A ≡ {*w*. *path A w* (*initial A*) ∧ *test* (*condition A*) *w* (*states A w* (*initial A*)) (*initial A*)}

lemma *language*[*intro*]:
assumes *path A w* (*initial A*) *test* (*condition A*) *w* (*states A w* (*initial A*)) (*initial A*)
shows *w* ∈ *language A*
⟨*proof*⟩
lemma *language-elim*[*elim*]:
assumes *w* ∈ *language A*
obtains *path A w* (*initial A*) *test* (*condition A*) *w* (*states A w* (*initial A*)) (*initial A*)
⟨*proof*⟩

lemma *language-alphabet*: *language A* ⊆ *lists* (*alphabet A*) ⟨*proof*⟩

end

locale *automaton-run* =
automaton automaton alphabet initial transition condition
for *automaton* :: *'label set* ⇒ *'state* ⇒ (*'label* ⇒ *'state* ⇒ *'state*) ⇒ *'condition* ⇒ *'automaton*
and *alphabet initial transition condition*
\+
fixes *test* :: *'condition* ⇒ *'label stream* ⇒ *'state stream* ⇒ *'state* ⇒ *bool*
begin

definition *language* :: *'automaton* ⇒ *'label stream set* **where**
language A ≡ {*w*. *run A w* (*initial A*) ∧ *test* (*condition A*) *w* (*trace A w* (*initial A*)) (*initial A*)}

lemma *language*[*intro*]:
assumes *run A w* (*initial A*) *test* (*condition A*) *w* (*trace A w* (*initial A*)) (*initial A*)
shows $w \in$ *language A*
⟨*proof*⟩
lemma *language-elim*[*elim*]:
assumes $w \in$ *language A*
obtains *run A w* (*initial A*) *test* (*condition A*) *w* (*trace A w* (*initial A*)) (*initial A*)
⟨*proof*⟩

lemma *language-alphabet*: *language A* $\subseteq$ *streams* (*alphabet A*) ⟨*proof*⟩

end

locale *automaton-degeneralization* =
a: *automaton* $automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$ +
b: *automaton* $automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$
for $automaton_1$ :: $'label$ *set* $\Rightarrow$ $'state$ $\Rightarrow$ ($'label \Rightarrow 'state \Rightarrow 'state$) $\Rightarrow$ $'item$ *pred gen* $\Rightarrow$ $'automaton_1$
and $alphabet_1$ $initial_1$ $transition_1$ $condition_1$
and $automaton_2$ :: $'label$ *set* $\Rightarrow$ $'state$ *degen* $\Rightarrow$ ($'label \Rightarrow 'state$ *degen* $\Rightarrow 'state$ *degen*) $\Rightarrow$ $'item$-*degen pred* $\Rightarrow$ $'automaton_2$
and $alphabet_2$ $initial_2$ $transition_2$ $condition_2$
+
fixes *item* :: $'state \times 'label \times 'state \Rightarrow 'item$
fixes *translate* :: $'item$-*degen* $\Rightarrow$ $'item$ *degen*
begin

definition *degeneralize* :: $'automaton_1 \Rightarrow 'automaton_2$ **where**
degeneralize A $\equiv$ $automaton_2$
($alphabet_1$ *A*)
($initial_1$ *A*, *0*)
(λ *a* (*p*, *k*). ($transition_1$ *A a p*, *count* ($condition_1$ *A*) (*item* (*p*, *a*, $transition_1$ *A a p*)) *k*))
(*degen* ($condition_1$ *A*) $\circ$ *translate*)

lemma *degeneralize-simps*[*simp*]:
$alphabet_2$ (*degeneralize A*) = $alphabet_1$ *A*
$initial_2$ (*degeneralize A*) = ($initial_1$ *A*, *0*)
$transition_2$ (*degeneralize A*) *a* (*p*, *k*) =
($transition_1$ *A a p*, *count* ($condition_1$ *A*) (*item* (*p*, *a*, $transition_1$ *A a p*)) *k*)
$condition_2$ (*degeneralize A*) = *degen* ($condition_1$ *A*) $\circ$ *translate*
⟨*proof*⟩

lemma *degeneralize-target*[*simp*]: *b.target* (*degeneralize A*) *w* (*p*, *k*) =
(*a.target A w p*, *fold* (*count* ($condition_1$ *A*) $\circ$ *item*) (*p* # *a.states A w p* || *w* || *a.states A w p*) *k*)

⟨*proof*⟩

lemma *degeneralize-states*[*simp*]: *b.states* (*degeneralize A*) *w* (*p*, *k*) = *a.states A w p* || *scan* (*count* ($condition_1$ *A*) ∘ *item*) (*p* # *a.states A w p* || *w* || *a.states A w p*) *k*

⟨*proof*⟩

lemma *degeneralize-trace*[*simp*]: *b.trace* (*degeneralize A*) *w* (*p*, *k*) = *a.trace A w p* ||| *sscan* (*count* ($condition_1$ *A*) ∘ *item*) (*p* ## *a.trace A w p* ||| *w* ||| *a.trace A w p*) *k*

⟨*proof*⟩

lemma *degeneralize-path*[*iff*]: *b.path* (*degeneralize A*) *w* (*p*, *k*) ⟷ *a.path A w p*

⟨*proof*⟩

lemma *degeneralize-run*[*iff*]: *b.run* (*degeneralize A*) *w* (*p*, *k*) ⟷ *a.run A w p*

⟨*proof*⟩

lemma *degeneralize-reachable-fst*[*simp*]: *fst* ‘ *b.reachable* (*degeneralize A*) (*p*, *k*) = *a.reachable A p*

⟨*proof*⟩

lemma *degeneralize-reachable-snd-empty*[*simp*]:
assumes $condition_1$ *A* = []
shows *snd* ‘ *b.reachable* (*degeneralize A*) (*p*, *k*) = {*k*}

⟨*proof*⟩

lemma *degeneralize-reachable-empty*[*simp*]:
assumes $condition_1$ *A* = []
shows *b.reachable* (*degeneralize A*) (*p*, *k*) = *a.reachable A p* × {*k*}

⟨*proof*⟩

lemma *degeneralize-reachable-snd*:
snd ‘ *b.reachable* (*degeneralize A*) (*p*, *k*) ⊆ *insert k* {*0* ..< *length* ($condition_1$ *A*)}

⟨*proof*⟩

lemma *degeneralize-reachable*:
b.reachable (*degeneralize A*) (*p*, *k*) ⊆ *a.reachable A p* × *insert k* {*0* ..< *length* ($condition_1$ *A*)}

⟨*proof*⟩

lemma *degeneralize-nodes-fst*[*simp*]: *fst* ‘ *b.nodes* (*degeneralize A*) = *a.nodes A*

⟨*proof*⟩

lemma *degeneralize-nodes-snd-empty*:
assumes $condition_1$ *A* = []
shows *snd* ‘ *b.nodes* (*degeneralize A*) = {*0*}

⟨*proof*⟩

lemma *degeneralize-nodes-empty*:
assumes $condition_1$ *A* = []
shows *b.nodes* (*degeneralize A*) = *a.nodes A* × {*0*}

⟨*proof*⟩

lemma *degeneralize-nodes-snd*:
snd ‘ *b.nodes* (*degeneralize A*) ⊆ *insert 0* {*0* ..< *length* ($condition_1$ *A*)}

⟨*proof*⟩

lemma *degeneralize-nodes*:
$b.nodes\ (degeneralize\ A) \subseteq a.nodes\ A \times insert\ 0\ \{0\ ..<\ length\ (condition_1\ A)\}$
$\langle proof \rangle$

lemma *degeneralize-nodes-finite*[*iff*]: $finite\ (b.nodes\ (degeneralize\ A)) \longleftrightarrow finite\ (a.nodes\ A)$
$\langle proof \rangle$
lemma *degeneralize-nodes-card*: $card\ (b.nodes\ (degeneralize\ A)) \leq$
$max\ 1\ (length\ (condition_1\ A)) * card\ (a.nodes\ A)$
$\langle proof \rangle$

end

locale *automaton-degeneralization-run* =
automaton-degeneralization
$automaton_1\ alphabet_1\ initial_1\ transition_1\ condition_1$
$automaton_2\ alphabet_2\ initial_2\ transition_2\ condition_2$
item translate +
a: $automaton\text{-}run\ automaton_1\ alphabet_1\ initial_1\ transition_1\ condition_1\ test_1$ +
b: $automaton\text{-}run\ automaton_2\ alphabet_2\ initial_2\ transition_2\ condition_2\ test_2$
for $automaton_1\ alphabet_1\ initial_1\ transition_1\ condition_1\ test_1$
and $automaton_2\ alphabet_2\ initial_2\ transition_2\ condition_2\ test_2$
and *item translate*
+
assumes *test*[*iff*]: $test_2\ (degen\ cs \circ translate)\ w$
$(r\ |||\ sscan\ (count\ cs \circ item)\ (p\ \#\#\ r\ |||\ w\ |||\ r)\ k)\ (p,\ k) \longleftrightarrow test_1\ cs\ w\ r\ p$
begin

lemma *degeneralize-language*[*simp*]: $b.language\ (degeneralize\ A) = a.language\ A$ $\langle proof \rangle$

end

locale *automaton-product* =
a: $automaton\ automaton_1\ alphabet_1\ initial_1\ transition_1\ condition_1$ +
b: $automaton\ automaton_2\ alphabet_2\ initial_2\ transition_2\ condition_2$ +
c: $automaton\ automaton_3\ alphabet_3\ initial_3\ transition_3\ condition_3$
for $automaton_1 :: 'label\ set \Rightarrow 'state_1 \Rightarrow ('label \Rightarrow 'state_1 \Rightarrow 'state_1) \Rightarrow 'condition_1 \Rightarrow 'automaton_1$
and $alphabet_1\ initial_1\ transition_1\ condition_1$
and $automaton_2 :: 'label\ set \Rightarrow 'state_2 \Rightarrow ('label \Rightarrow 'state_2 \Rightarrow 'state_2) \Rightarrow 'condition_2 \Rightarrow 'automaton_2$
and $alphabet_2\ initial_2\ transition_2\ condition_2$
and $automaton_3 :: 'label\ set \Rightarrow 'state_1 \times 'state_2 \Rightarrow ('label \Rightarrow 'state_1 \times 'state_2 \Rightarrow 'state_1 \times 'state_2) \Rightarrow 'condition_3 \Rightarrow 'automaton_3$
and $alphabet_3\ initial_3\ transition_3\ condition_3$
+
fixes $condition :: 'condition_1 \Rightarrow 'condition_2 \Rightarrow 'condition_3$

begin

definition *product* :: $'automaton_1 \Rightarrow 'automaton_2 \Rightarrow 'automaton_3$ **where**
$product\ A\ B \equiv automaton_3$
$(alphabet_1\ A \cap alphabet_2\ B)$
$(initial_1\ A,\ initial_2\ B)$
$(\lambda\ a\ (p,\ q).\ (transition_1\ A\ a\ p,\ transition_2\ B\ a\ q))$
$(condition\ (condition_1\ A)\ (condition_2\ B))$

lemma *product-simps*[*simp*]:
$alphabet_3\ (product\ A\ B) = alphabet_1\ A \cap alphabet_2\ B$
$initial_3\ (product\ A\ B) = (initial_1\ A,\ initial_2\ B)$
$transition_3\ (product\ A\ B)\ a\ (p,\ q) = (transition_1\ A\ a\ p,\ transition_2\ B\ a\ q)$
$condition_3\ (product\ A\ B) = condition\ (condition_1\ A)\ (condition_2\ B)$
⟨*proof*⟩

lemma *product-target*[*simp*]: $c.target\ (product\ A\ B)\ w\ (p,\ q) = (a.target\ A\ w\ p,\ b.target\ B\ w\ q)$
⟨*proof*⟩
lemma *product-states*[*simp*]: $c.states\ (product\ A\ B)\ w\ (p,\ q) = a.states\ A\ w\ p\ ||\ b.states\ B\ w\ q$
⟨*proof*⟩
lemma *product-trace*[*simp*]: $c.trace\ (product\ A\ B)\ w\ (p,\ q) = a.trace\ A\ w\ p\ |||\ b.trace\ B\ w\ q$
⟨*proof*⟩

lemma *product-path*[*iff*]: $c.path\ (product\ A\ B)\ w\ (p,\ q) \longleftrightarrow a.path\ A\ w\ p \wedge b.path\ B\ w\ q$
⟨*proof*⟩
lemma *product-run*[*iff*]: $c.run\ (product\ A\ B)\ w\ (p,\ q) \longleftrightarrow a.run\ A\ w\ p \wedge b.run\ B\ w\ q$
⟨*proof*⟩

lemma *product-reachable*[*simp*]: $c.reachable\ (product\ A\ B)\ (p,\ q) \subseteq a.reachable\ A\ p \times b.reachable\ B\ q$
⟨*proof*⟩
lemma *product-nodes*[*simp*]: $c.nodes\ (product\ A\ B) \subseteq a.nodes\ A \times b.nodes\ B$
⟨*proof*⟩
lemma *product-reachable-fst*[*simp*]:
assumes $alphabet_1\ A \subseteq alphabet_2\ B$
shows $fst\ `\ c.reachable\ (product\ A\ B)\ (p,\ q) = a.reachable\ A\ p$
⟨*proof*⟩
lemma *product-reachable-snd*[*simp*]:
assumes $alphabet_1\ A \supseteq alphabet_2\ B$
shows $snd\ `\ c.reachable\ (product\ A\ B)\ (p,\ q) = b.reachable\ B\ q$
⟨*proof*⟩
lemma *product-nodes-fst*[*simp*]:
assumes $alphabet_1\ A \subseteq alphabet_2\ B$
shows $fst\ `\ c.nodes\ (product\ A\ B) = a.nodes\ A$

$\langle proof \rangle$
lemma *product-nodes-snd*[*simp*]:
assumes $alphabet_1\ A \supseteq alphabet_2\ B$
shows *snd* ' *c.nodes* (*product* *A* *B*) = *b.nodes* *B*
$\langle proof \rangle$

lemma *product-nodes-finite*[*intro*]:
assumes *finite* (*a.nodes* *A*) *finite* (*b.nodes* *B*)
shows *finite* (*c.nodes* (*product* *A* *B*))
$\langle proof \rangle$
lemma *product-nodes-finite-strong*[*iff*]:
assumes $alphabet_1\ A = alphabet_2\ B$
shows *finite* (*c.nodes* (*product* *A* *B*)) $\longleftrightarrow$ *finite* (*a.nodes* *A*) $\wedge$ *finite* (*b.nodes* *B*)
$\langle proof \rangle$
lemma *product-nodes-card*[*intro*]:
assumes *finite* (*a.nodes* *A*) *finite* (*b.nodes* *B*)
shows *card* (*c.nodes* (*product* *A* *B*)) $\leq$ *card* (*a.nodes* *A*) $*$ *card* (*b.nodes* *B*)
$\langle proof \rangle$
lemma *product-nodes-card-strong*[*intro*]:
assumes $alphabet_1\ A = alphabet_2\ B$
shows *card* (*c.nodes* (*product* *A* *B*)) $\leq$ *card* (*a.nodes* *A*) $*$ *card* (*b.nodes* *B*)
$\langle proof \rangle$

end

locale *automaton-intersection-path* =
automaton-product
$automaton_1\ alphabet_1\ initial_1\ transition_1\ condition_1$
$automaton_2\ alphabet_2\ initial_2\ transition_2\ condition_2$
$automaton_3\ alphabet_3\ initial_3\ transition_3\ condition_3$
condition +
a: *automaton-path* $automaton_1\ alphabet_1\ initial_1\ transition_1\ condition_1\ test_1$ +
b: *automaton-path* $automaton_2\ alphabet_2\ initial_2\ transition_2\ condition_2\ test_2$ +
c: *automaton-path* $automaton_3\ alphabet_3\ initial_3\ transition_3\ condition_3\ test_3$
for $automaton_1\ alphabet_1\ initial_1\ transition_1\ condition_1\ test_1$
and $automaton_2\ alphabet_2\ initial_2\ transition_2\ condition_2\ test_2$
and $automaton_3\ alphabet_3\ initial_3\ transition_3\ condition_3\ test_3$
and *condition*
+
assumes *test*[*iff*]: *length* r = *length* $s \Longrightarrow$
$test_3\ (condition\ c_1\ c_2)\ w\ (r \mid\mid s)\ (p,\ q) \longleftrightarrow test_1\ c_1\ w\ r\ p \wedge test_2\ c_2\ w\ s\ q$
begin

lemma *product-language*[*simp*]: *c.language* (*product* *A* *B*) = *a.language* *A* $\cap$ *b.language* *B* $\langle proof \rangle$

end

locale *automaton-union-path* =
automaton-product
$automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$
$automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$
$automaton_3$ $alphabet_3$ $initial_3$ $transition_3$ $condition_3$
condition +
a: *automaton-path* $automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$ $test_1$ +
b: *automaton-path* $automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$ $test_2$ +
c: *automaton-path* $automaton_3$ $alphabet_3$ $initial_3$ $transition_3$ $condition_3$ $test_3$
for $automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$ $test_1$
and $automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$ $test_2$
and $automaton_3$ $alphabet_3$ $initial_3$ $transition_3$ $condition_3$ $test_3$
and *condition*
+
assumes *test*[*iff*]: *length* r = *length* $s \Longrightarrow$
$test_3$ (*condition* c_1 c_2) w (r || s) (p, q) $\longleftrightarrow$ $test_1$ c_1 w r p $\vee$ $test_2$ c_2 w s q
begin

lemma *product-language*[*simp*]:
assumes $alphabet_1$ A = $alphabet_2$ B
shows *c.language* (*product* A B) = *a.language* A $\cup$ *b.language* B
⟨*proof*⟩

end

locale *automaton-intersection-run* =
automaton-product
$automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$
$automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$
$automaton_3$ $alphabet_3$ $initial_3$ $transition_3$ $condition_3$
condition +
a: *automaton-run* $automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$ $test_1$ +
b: *automaton-run* $automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$ $test_2$ +
c: *automaton-run* $automaton_3$ $alphabet_3$ $initial_3$ $transition_3$ $condition_3$ $test_3$
for $automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$ $test_1$
and $automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$ $test_2$
and $automaton_3$ $alphabet_3$ $initial_3$ $transition_3$ $condition_3$ $test_3$
and *condition*
+
assumes *test*[*iff*]: $test_3$ (*condition* c_1 c_2) w (r ||| s) (p, q) $\longleftrightarrow$ $test_1$ c_1 w r p $\wedge$ $test_2$ c_2 w s q
begin

lemma *product-language*[*simp*]: *c.language* (*product* A B) = *a.language* A $\cap$ *b.language* B ⟨*proof*⟩

end

locale *automaton-union-run* =

automaton-product
  $automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$
  $automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$
  $automaton_3$ $alphabet_3$ $initial_3$ $transition_3$ $condition_3$
  condition +
a: *automaton-run* $automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$ $test_1$ +
b: *automaton-run* $automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$ $test_2$ +
c: *automaton-run* $automaton_3$ $alphabet_3$ $initial_3$ $transition_3$ $condition_3$ $test_3$
for $automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$ $test_1$
and $automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$ $test_2$
and $automaton_3$ $alphabet_3$ $initial_3$ $transition_3$ $condition_3$ $test_3$
and *condition*
+
assumes *test*[*iff*]: $test_3$ (*condition* c_1 c_2) *w* (*r* ||| *s*) (*p*, *q*) $\longleftrightarrow$ $test_1$ c_1 *w r p* $\vee$ $test_2$ c_2 *w s q*
begin

lemma *product-language*[*simp*]:
  assumes $alphabet_1$ *A* = $alphabet_2$ *B*
  shows *c.language* (*product A B*) = *a.language A* $\cup$ *b.language B*
  ⟨*proof*⟩

end

locale *automaton-product-list* =
  a: *automaton* $automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$ +
  b: *automaton* $automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$
for $automaton_1$ :: *'label set* $\Rightarrow$ *'state* $\Rightarrow$ (*'label* $\Rightarrow$ *'state* $\Rightarrow$ *'state*) $\Rightarrow$ $'condition_1$ $\Rightarrow$ $'automaton_1$
and $alphabet_1$ $initial_1$ $transition_1$ $condition_1$
and $automaton_2$ :: *'label set* $\Rightarrow$ *'state list* $\Rightarrow$ (*'label* $\Rightarrow$ *'state list* $\Rightarrow$ *'state list*) $\Rightarrow$ $'condition_2$ $\Rightarrow$ $'automaton_2$
and $alphabet_2$ $initial_2$ $transition_2$ $condition_2$
+
fixes *condition* :: $'condition_1$ *list* $\Rightarrow$ $'condition_2$
begin

definition *product* :: $'automaton_1$ *list* $\Rightarrow$ $'automaton_2$ **where**
  product AA $\equiv$ $automaton_2$
    ($\bigcap$ ($alphabet_1$ ‘ *set AA*))
    (*map* $initial_1$ *AA*)
    (λ *a ps*. *map2* (λ *A p*. $transition_1$ *A a p*) *AA ps*)
    (*condition* (*map* $condition_1$ *AA*))

lemma *product-simps*[*simp*]:
  $alphabet_2$ (*product AA*) = $\bigcap$ ($alphabet_1$ ‘ *set AA*)
  $initial_2$ (*product AA*) = *map* $initial_1$ *AA*
  $transition_2$ (*product AA*) *a ps* = *map2* (λ *A p*. $transition_1$ *A a p*) *AA ps*

$condition_2$ $(product\ AA) = condition\ (map\ condition_1\ AA)$
⟨proof⟩

lemma *product-trace-smap*:
assumes $length\ ps = length\ AA\ k < length\ AA$
shows $smap\ (\lambda\ ps.\ ps\ !\ k)\ (b.trace\ (product\ AA)\ w\ ps) = a.trace\ (AA\ !\ k)\ w\ (ps\ !\ k)$
⟨proof⟩

lemma *product-nodes*: $b.nodes\ (product\ AA) \subseteq listset\ (map\ a.nodes\ AA)$
⟨proof⟩

lemma *product-nodes-finite*[*intro*]:
assumes $list\text{-}all\ (finite \circ a.nodes)\ AA$
shows $finite\ (b.nodes\ (product\ AA))$
⟨proof⟩
lemma *product-nodes-card*:
assumes $list\text{-}all\ (finite \circ a.nodes)\ AA$
shows $card\ (b.nodes\ (product\ AA)) \leq prod\text{-}list\ (map\ (card \circ a.nodes)\ AA)$
⟨proof⟩

end

locale *automaton-intersection-list-run* =
automaton-product-list
$automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$
$automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$
condition +
a: *automaton-run* $automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$ $test_1$ +
b: *automaton-run* $automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$ $test_2$
for $automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$ $test_1$
and $automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$ $test_2$
and *condition*
+
assumes *test*[*iff*]: $test_2\ (condition\ cs)\ w\ rs\ ps \longleftrightarrow$
$(\forall\ k < length\ cs.\ test_1\ (cs\ !\ k)\ w\ (smap\ (\lambda\ ps.\ ps\ !\ k)\ rs)\ (ps\ !\ k))$
begin

lemma *product-language*[*simp*]: $b.language\ (product\ AA) = \bigcap\ (a.language\ `\ set\ AA)$
⟨proof⟩

end

locale *automaton-union-list-run* =
automaton-product-list
$automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$
$automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$

condition +
a: *automaton-run automaton*$_1$ *alphabet*$_1$ *initial*$_1$ *transition*$_1$ *condition*$_1$ *test*$_1$ +
b: *automaton-run automaton*$_2$ *alphabet*$_2$ *initial*$_2$ *transition*$_2$ *condition*$_2$ *test*$_2$
for *automaton*$_1$ *alphabet*$_1$ *initial*$_1$ *transition*$_1$ *condition*$_1$ *test*$_1$
and *automaton*$_2$ *alphabet*$_2$ *initial*$_2$ *transition*$_2$ *condition*$_2$ *test*$_2$
and *condition*
+
assumes *test*[*iff*]: *test*$_2$ (*condition cs*) *w rs ps* $\longleftrightarrow$
($\exists$ k < *length cs*. *test*$_1$ (*cs* ! *k*) *w* (*smap* (λ *ps*. *ps* ! *k*) *rs*) (*ps* ! *k*))
begin

lemma *product-language*[*simp*]:
assumes $\bigcap$ (*alphabet*$_1$ ' *set AA*) = $\bigcup$ (*alphabet*$_1$ ' *set AA*)
shows *b.language* (*product AA*) = $\bigcup$ (*a.language* ' *set AA*)
⟨*proof*⟩

end

locale *automaton-complement* =
a: *automaton automaton*$_1$ *alphabet*$_1$ *initial*$_1$ *transition*$_1$ *condition*$_1$ +
b: *automaton automaton*$_2$ *alphabet*$_2$ *initial*$_2$ *transition*$_2$ *condition*$_2$
for *automaton*$_1$ :: '*label set* $\Rightarrow$ '*state* $\Rightarrow$ ('*label* $\Rightarrow$ '*state* $\Rightarrow$ '*state*) $\Rightarrow$ '*condition*$_1$ $\Rightarrow$ '*automaton*$_1$
and *alphabet*$_1$ *initial*$_1$ *transition*$_1$ *condition*$_1$
and *automaton*$_2$ :: '*label set* $\Rightarrow$ '*state* $\Rightarrow$ ('*label* $\Rightarrow$ '*state* $\Rightarrow$ '*state*) $\Rightarrow$ '*condition*$_2$ $\Rightarrow$ '*automaton*$_2$
and *alphabet*$_2$ *initial*$_2$ *transition*$_2$ *condition*$_2$
+
fixes *condition* :: '*condition*$_1$ $\Rightarrow$ '*condition*$_2$
begin

definition *complement* :: '*automaton*$_1$ $\Rightarrow$ '*automaton*$_2$ **where**
complement A $\equiv$ *automaton*$_2$ (*alphabet*$_1$ *A*) (*initial*$_1$ *A*) (*transition*$_1$ *A*) (*condition* (*condition*$_1$ *A*))

lemma *combine-simps*[*simp*]:
alphabet$_2$ (*complement A*) = *alphabet*$_1$ *A*
initial$_2$ (*complement A*) = *initial*$_1$ *A*
transition$_2$ (*complement A*) = *transition*$_1$ *A*
condition$_2$ (*complement A*) = *condition* (*condition*$_1$ *A*)
⟨*proof*⟩

end

locale *automaton-complement-path* =
automaton-complement
automaton$_1$ *alphabet*$_1$ *initial*$_1$ *transition*$_1$ *condition*$_1$
automaton$_2$ *alphabet*$_2$ *initial*$_2$ *transition*$_2$ *condition*$_2$
condition +

a: *automaton-path automaton*$_1$ *alphabet*$_1$ *initial*$_1$ *transition*$_1$ *condition*$_1$ *test*$_1$ +
b: *automaton-path automaton*$_2$ *alphabet*$_2$ *initial*$_2$ *transition*$_2$ *condition*$_2$ *test*$_2$
for *automaton*$_1$ *alphabet*$_1$ *initial*$_1$ *transition*$_1$ *condition*$_1$ *test*$_1$
and *automaton*$_2$ *alphabet*$_2$ *initial*$_2$ *transition*$_2$ *condition*$_2$ *test*$_2$
and *condition*
+
assumes *test*[*iff*]: *test*$_2$ (*condition c*) *w r p* $\longleftrightarrow$ $\neg$ *test*$_1$ *c w r p*
begin

lemma *complement-language*[*simp*]: *b.language* (*complement A*) = *lists* (*alphabet*$_1$ *A*) − *a.language A*
⟨*proof*⟩

end

locale *automaton-complement-run* =
automaton-complement
automaton$_1$ *alphabet*$_1$ *initial*$_1$ *transition*$_1$ *condition*$_1$
automaton$_2$ *alphabet*$_2$ *initial*$_2$ *transition*$_2$ *condition*$_2$
condition +
a: *automaton-run automaton*$_1$ *alphabet*$_1$ *initial*$_1$ *transition*$_1$ *condition*$_1$ *test*$_1$ +
b: *automaton-run automaton*$_2$ *alphabet*$_2$ *initial*$_2$ *transition*$_2$ *condition*$_2$ *test*$_2$
for *automaton*$_1$ *alphabet*$_1$ *initial*$_1$ *transition*$_1$ *condition*$_1$ *test*$_1$
and *automaton*$_2$ *alphabet*$_2$ *initial*$_2$ *transition*$_2$ *condition*$_2$ *test*$_2$
and *condition*
+
assumes *test*[*iff*]: *test*$_2$ (*condition c*) *w r p* $\longleftrightarrow$ $\neg$ *test*$_1$ *c w r p*
begin

lemma *complement-language*[*simp*]: *b.language* (*complement A*) = *streams* (*alphabet*$_1$ *A*) − *a.language A*
⟨*proof*⟩

end

end

11 Deterministic Finite Automata

theory *DFA*
imports ../*Deterministic*
begin

datatype ('*label*, '*state*) *dfa* = *dfa*
(*alphabet*: '*label set*)
(*initial*: '*state*)
(*transition*: '*label* ⇒ '*state* ⇒ '*state*)
(*accepting*: '*state pred*)

global-interpretation *dfa*: *automaton dfa alphabet initial transition accepting*
defines *path* = *dfa.path* **and** *run* = *dfa.run* **and** *reachable* = *dfa.reachable* **and** *nodes* = *dfa.nodes*
⟨*proof*⟩
global-interpretation *dfa*: *automaton-path dfa alphabet initial transition accepting* λ *P w r p. P* (*last* (*p* # *r*))
defines *language* = *dfa.language*
⟨*proof*⟩

abbreviation *target* **where** *target* ≡ *dfa.target*
abbreviation *states* **where** *states* ≡ *dfa.states*
abbreviation *trace* **where** *trace* ≡ *dfa.trace*
abbreviation *successors* **where** *successors* ≡ *dfa.successors TYPE*(*'label*)

global-interpretation *intersection*: *automaton-intersection-path*
dfa alphabet initial transition accepting λ *P w r p. P* (*last* (*p* # *r*))
dfa alphabet initial transition accepting λ *P w r p. P* (*last* (*p* # *r*))
dfa alphabet initial transition accepting λ *P w r p. P* (*last* (*p* # *r*))
λ c_1 c_2 (*p*, *q*). c_1 *p* $\wedge$ c_2 *q*
defines *intersect* = *intersection.product*
⟨*proof*⟩

global-interpretation *union*: *automaton-union-path*
dfa alphabet initial transition accepting λ *P w r p. P* (*last* (*p* # *r*))
dfa alphabet initial transition accepting λ *P w r p. P* (*last* (*p* # *r*))
dfa alphabet initial transition accepting λ *P w r p. P* (*last* (*p* # *r*))
λ c_1 c_2 (*p*, *q*). c_1 *p* $\vee$ c_2 *q*
defines *union* = *union.product*
⟨*proof*⟩

global-interpretation *complement*: *automaton-complement-path*
dfa alphabet initial transition accepting λ *P w r p. P* (*last* (*p* # *r*))
dfa alphabet initial transition accepting λ *P w r p. P* (*last* (*p* # *r*))
λ *c p.* ¬ *c p*
defines *complement* = *complement.complement*
⟨*proof*⟩

end

12 Nondeterministic Automata

theory *Nondeterministic*
imports
../*Transition-Systems*/*Transition-System*
../*Transition-Systems*/*Transition-System-Extra*
../*Transition-Systems*/*Transition-System-Construction*
../*Basic*/*Degeneralization*
begin

locale *automaton* =
  fixes *automaton* :: $'label\ set \Rightarrow 'state\ set \Rightarrow ('label \Rightarrow 'state \Rightarrow 'state\ set) \Rightarrow 'condition \Rightarrow 'automaton$
  fixes *alphabet initial transition condition*
  assumes *automaton*[*simp*]: *automaton* (*alphabet A*) (*initial A*) (*transition A*) (*condition A*) = *A*
  assumes *alphabet*[*simp*]: *alphabet* (*automaton a i t c*) = *a*
  assumes *initial*[*simp*]: *initial* (*automaton a i t c*) = *i*
  assumes *transition*[*simp*]: *transition* (*automaton a i t c*) = *t*
  assumes *condition*[*simp*]: *condition* (*automaton a i t c*) = *c*
begin

  sublocale *transition-system-initial*
    λ *a p. snd a* λ *a p. fst a* $\in$ *alphabet A* $\wedge$ *snd a* $\in$ *transition A* (*fst a*) *p* λ *p. p* $\in$ *initial A*
    for *A*
    defines *path'* = *path* **and** *run'* = *run* **and** *reachable'* = *reachable* **and** *nodes'* = *nodes*
    ⟨*proof*⟩

  lemma *states-alt-def*: *states r p* = *map snd r* ⟨*proof*⟩
  lemma *trace-alt-def*: *trace r p* = *smap snd r* ⟨*proof*⟩

  lemma *successors-alt-def*: *successors A p* = ($\bigcup$ *a* $\in$ *alphabet A*. *transition A a p*) ⟨*proof*⟩

  lemma *reachable-transition*[*intro*]:
    assumes *a* $\in$ *alphabet A q* $\in$ *reachable A p r* $\in$ *transition A a q*
    shows *r* $\in$ *reachable A p*
    ⟨*proof*⟩
  lemma *nodes-transition*[*intro*]:
    assumes *a* $\in$ *alphabet A p* $\in$ *nodes A q* $\in$ *transition A a p*
    shows *q* $\in$ *nodes A*
    ⟨*proof*⟩

  lemma *path-alphabet*:
    assumes *length r* = *length w path A* (*w* || *r*) *p*
    shows *w* $\in$ *lists* (*alphabet A*)
    ⟨*proof*⟩
  lemma *run-alphabet*:
    assumes *run A* (*w* ||| *r*) *p*
    shows *w* $\in$ *streams* (*alphabet A*)
    ⟨*proof*⟩

  definition *restrict* :: $'automaton \Rightarrow 'automaton$ **where**
    restrict A $\equiv$ *automaton*
      (*alphabet A*)
      (*initial A*)
      (λ *a p. if a* $\in$ *alphabet A then transition A a p else* {})

$(condition\ A)$

lemma *restrict-simps*[*simp*]:
$alphabet\ (restrict\ A) = alphabet\ A$
$initial\ (restrict\ A) = initial\ A$
$transition\ (restrict\ A)\ a\ p = (if\ a \in alphabet\ A\ then\ transition\ A\ a\ p\ else\ \{\})$
$condition\ (restrict\ A) = condition\ A$
⟨*proof*⟩

lemma *restrict-path*[*simp*]: $path\ (restrict\ A) = path\ A$
⟨*proof*⟩
lemma *restrict-run*[*simp*]: $run\ (restrict\ A) = run\ A$
⟨*proof*⟩

end

locale *automaton-path* =
automaton automaton alphabet initial transition condition
for $automaton :: 'label\ set \Rightarrow 'state\ set \Rightarrow ('label \Rightarrow 'state \Rightarrow 'state\ set) \Rightarrow 'condition \Rightarrow 'automaton$
and *alphabet initial transition condition*
+
fixes $test :: 'condition \Rightarrow 'label\ list \Rightarrow 'state\ list \Rightarrow 'state \Rightarrow bool$
begin

definition $language :: 'automaton \Rightarrow 'label\ list\ set$ **where**
$language\ A \equiv \{w\ |w\ r\ p.\ length\ r = length\ w \wedge p \in initial\ A \wedge path\ A\ (w\ ||\ r)\ p \wedge test\ (condition\ A)\ w\ r\ p\}$

lemma *language*[*intro*]:
assumes $length\ r = length\ w\ p \in initial\ A\ path\ A\ (w\ ||\ r)\ p\ test\ (condition\ A)\ w\ r\ p$
shows $w \in language\ A$
⟨*proof*⟩
lemma *language-elim*[*elim*]:
assumes $w \in language\ A$
obtains $r\ p$
where $length\ r = length\ w\ p \in initial\ A\ path\ A\ (w\ ||\ r)\ p\ test\ (condition\ A)\ w\ r\ p$
⟨*proof*⟩

lemma *language-alphabet*: $language\ A \subseteq lists\ (alphabet\ A)$ ⟨*proof*⟩

lemma *restrict-language*[*simp*]: $language\ (restrict\ A) = language\ A$ ⟨*proof*⟩

end

locale *automaton-run* =
automaton automaton alphabet initial transition condition

for *automaton* :: *'label set* ⇒ *'state set* ⇒ (*'label* ⇒ *'state* ⇒ *'state set*) ⇒ *'condition* ⇒ *'automaton*
and *alphabet initial transition condition*
+
fixes *test* :: *'condition* ⇒ *'label stream* ⇒ *'state stream* ⇒ *'state* ⇒ *bool*
begin

definition *language* :: *'automaton* ⇒ *'label stream set* **where**
language A ≡ {*w* |*w r p*. *p* ∈ *initial A* ∧ *run A* (*w* ||| *r*) *p* ∧ *test* (*condition A*) *w r p*}

lemma *language*[*intro*]:
assumes *p* ∈ *initial A run A* (*w* ||| *r*) *p test* (*condition A*) *w r p*
shows *w* ∈ *language A*
⟨*proof*⟩
lemma *language-elim*[*elim*]:
assumes *w* ∈ *language A*
obtains *r p*
where *p* ∈ *initial A run A* (*w* ||| *r*) *p test* (*condition A*) *w r p*
⟨*proof*⟩

lemma *language-alphabet*: *language A* ⊆ *streams* (*alphabet A*) ⟨*proof*⟩

lemma *restrict-language*[*simp*]: *language* (*restrict A*) = *language A* ⟨*proof*⟩

end

locale *automaton-degeneralization* =
a: *automaton* $automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$ +
b: *automaton* $automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$
for $automaton_1$ :: *'label set* ⇒ *'state set* ⇒ (*'label* ⇒ *'state* ⇒ *'state set*) ⇒ *'item pred gen* ⇒ $'automaton_1$
and $alphabet_1$ $initial_1$ $transition_1$ $condition_1$
and $automaton_2$ :: *'label set* ⇒ *'state degen set* ⇒ (*'label* ⇒ *'state degen* ⇒ *'state degen set*) ⇒ *'item-degen pred* ⇒ $'automaton_2$
and $alphabet_2$ $initial_2$ $transition_2$ $condition_2$
+
fixes *item* :: *'state* × *'label* × *'state* ⇒ *'item*
fixes *translate* :: *'item-degen* ⇒ *'item degen*
begin

definition *degeneralize* :: $'automaton_1$ ⇒ $'automaton_2$ **where**
degeneralize A ≡ $automaton_2$
($alphabet_1$ *A*)
($initial_1$ *A* × {*0*})
(λ *a* (*p*, *k*). {(*q*, *count* ($condition_1$ *A*) (*item* (*p*, *a*, *q*)) *k*) |*q*. *q* ∈ $transition_1$ *A a p*})
(*degen* ($condition_1$ *A*) ∘ *translate*)

lemma *degeneralize-simps*[*simp*]:
alphabet$_2$ (*degeneralize A*) = *alphabet*$_1$ *A*
initial$_2$ (*degeneralize A*) = *initial*$_1$ *A* × {*0*}
transition$_2$ (*degeneralize A*) *a* (*p*, *k*) =
{(*q*, *count* (*condition*$_1$ *A*) (*item* (*p*, *a*, *q*)) *k*) |*q*. *q* ∈ *transition*$_1$ *A a p*}
condition$_2$ (*degeneralize A*) = *degen* (*condition*$_1$ *A*) ∘ *translate*
⟨*proof*⟩

lemma *run-degeneralize*:
assumes *a.run A* (*w* ||| *r*) *p*
shows *b.run* (*degeneralize A*) (*w* ||| *r* ||| *sscan* (*count* (*condition*$_1$ *A*) ∘ *item*) (*p* ## *r* ||| *w* ||| *r*) *k*) (*p*, *k*)
⟨*proof*⟩
lemma *degeneralize-run*:
assumes *b.run* (*degeneralize A*) (*w* ||| *rs*) *pk*
obtains *r s p k*
where *rs* = *r* ||| *s pk* = (*p*, *k*) *a.run A* (*w* ||| *r*) *p s* = *sscan* (*count* (*condition*$_1$ *A*) ∘ *item*) (*p* ## *r* ||| *w* ||| *r*) *k*
⟨*proof*⟩

lemma *degeneralize-nodes*:
b.nodes (*degeneralize A*) ⊆ *a.nodes A* × *insert 0* {*0* ..< *length* (*condition*$_1$ *A*)}
⟨*proof*⟩
lemma *nodes-degeneralize*: *a.nodes A* ⊆ *fst* ‘ *b.nodes* (*degeneralize A*)
⟨*proof*⟩

lemma *degeneralize-nodes-finite*[*iff*]: *finite* (*b.nodes* (*degeneralize A*)) ⟷ *finite* (*a.nodes A*)
⟨*proof*⟩

end

locale *automaton-degeneralization-run* =
automaton-degeneralization
automaton$_1$ *alphabet*$_1$ *initial*$_1$ *transition*$_1$ *condition*$_1$
automaton$_2$ *alphabet*$_2$ *initial*$_2$ *transition*$_2$ *condition*$_2$
item translate +
a: *automaton-run automaton*$_1$ *alphabet*$_1$ *initial*$_1$ *transition*$_1$ *condition*$_1$ *test*$_1$ +
b: *automaton-run automaton*$_2$ *alphabet*$_2$ *initial*$_2$ *transition*$_2$ *condition*$_2$ *test*$_2$
for *automaton*$_1$ *alphabet*$_1$ *initial*$_1$ *transition*$_1$ *condition*$_1$ *test*$_1$
and *automaton*$_2$ *alphabet*$_2$ *initial*$_2$ *transition*$_2$ *condition*$_2$ *test*$_2$
and *item translate*
+
assumes *test*[*iff*]: *test*$_2$ (*degen cs* ∘ *translate*) *w*
(*r* ||| *sscan* (*count cs* ∘ *item*) (*p* ## *r* ||| *w* ||| *r*) *k*) (*p*, *k*) ⟷ *test*$_1$ *cs w r p*
begin

lemma *degeneralize-language*[*simp*]: *b.language* (*degeneralize A*) = *a.language*

A
⟨*proof*⟩

end

locale *automaton-product* =
a: *automaton automaton*$_1$ *alphabet*$_1$ *initial*$_1$ *transition*$_1$ *condition*$_1$ +
b: *automaton automaton*$_2$ *alphabet*$_2$ *initial*$_2$ *transition*$_2$ *condition*$_2$ +
c: *automaton automaton*$_3$ *alphabet*$_3$ *initial*$_3$ *transition*$_3$ *condition*$_3$
for $automaton_1$ :: $'label\ set \Rightarrow 'state_1\ set \Rightarrow ('label \Rightarrow 'state_1 \Rightarrow 'state_1\ set) \Rightarrow 'condition_1 \Rightarrow 'automaton_1$
and $alphabet_1\ initial_1\ transition_1\ condition_1$
and $automaton_2$ :: $'label\ set \Rightarrow 'state_2\ set \Rightarrow ('label \Rightarrow 'state_2 \Rightarrow 'state_2\ set) \Rightarrow 'condition_2 \Rightarrow 'automaton_2$
and $alphabet_2\ initial_2\ transition_2\ condition_2$
and $automaton_3$ :: $'label\ set \Rightarrow ('state_1 \times 'state_2)\ set \Rightarrow ('label \Rightarrow 'state_1 \times 'state_2 \Rightarrow ('state_1 \times 'state_2)\ set) \Rightarrow 'condition_3 \Rightarrow 'automaton_3$
and $alphabet_3\ initial_3\ transition_3\ condition_3$
+
fixes *condition* :: $'condition_1 \Rightarrow 'condition_2 \Rightarrow 'condition_3$
begin

definition *product* :: $'automaton_1 \Rightarrow 'automaton_2 \Rightarrow 'automaton_3$ **where**
product A B $\equiv$ $automaton_3$
$(alphabet_1\ A \cap alphabet_2\ B)$
$(initial_1\ A \times initial_2\ B)$
$(\lambda\ a\ (p, q).\ transition_1\ A\ a\ p \times transition_2\ B\ a\ q)$
$(condition\ (condition_1\ A)\ (condition_2\ B))$

lemma *product-simps*[*simp*]:
$alphabet_3\ (product\ A\ B) = alphabet_1\ A \cap alphabet_2\ B$
$initial_3\ (product\ A\ B) = initial_1\ A \times initial_2\ B$
$transition_3\ (product\ A\ B)\ a\ (p, q) = transition_1\ A\ a\ p \times transition_2\ B\ a\ q$
$condition_3\ (product\ A\ B) = condition\ (condition_1\ A)\ (condition_2\ B)$
⟨*proof*⟩

lemma *product-target*[*simp*]:
assumes *length w* = *length r length r* = *length s*
shows *c.target* (*w* || *r* || *s*) (*p*, *q*) = (*a.target* (*w* || *r*) *p*, *b.target* (*w* || *s*) *q*)
⟨*proof*⟩

lemma *product-path*[*iff*]:
assumes *length w* = *length r length r* = *length s*
shows *c.path* (*product A B*) (*w* || *r* || *s*) (*p*, *q*) $\longleftrightarrow$
a.path A (*w* || *r*) *p* $\wedge$ *b.path B* (*w* || *s*) *q*
⟨*proof*⟩
lemma *product-run*[*iff*]: *c.run* (*product A B*) (*w* ||| *r* ||| *s*) (*p*, *q*) $\longleftrightarrow$
a.run A (*w* ||| *r*) *p* $\wedge$ *b.run B* (*w* ||| *s*) *q*
⟨*proof*⟩

lemma *product-nodes*: *c.nodes* (*product* A B) $\subseteq$ *a.nodes* A $\times$ *b.nodes* B
⟨*proof*⟩

lemma *product-nodes-finite*[*intro*]:
assumes *finite* (*a.nodes* A) *finite* (*b.nodes* B)
shows *finite* (*c.nodes* (*product* A B))
⟨*proof*⟩

end

locale *automaton-intersection-path* =
automaton-product
$automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$
$automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$
$automaton_3$ $alphabet_3$ $initial_3$ $transition_3$ $condition_3$
condition +
a: *automaton-path* $automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$ $test_1$ +
b: *automaton-path* $automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$ $test_2$ +
c: *automaton-path* $automaton_3$ $alphabet_3$ $initial_3$ $transition_3$ $condition_3$ $test_3$
for $automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$ $test_1$
and $automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$ $test_2$
and $automaton_3$ $alphabet_3$ $initial_3$ $transition_3$ $condition_3$ $test_3$
and *condition*
+
assumes *test*[*iff*]: *length* r = *length* $w \Longrightarrow$ *length* s = *length* $w \Longrightarrow$
$test_3$ (*condition* c_1 c_2) w (r || s) (p, q) $\longleftrightarrow$ $test_1$ c_1 w r p $\wedge$ $test_2$ c_2 w s q
begin

lemma *product-language*[*simp*]: *c.language* (*product* A B) = *a.language* A $\cap$ *b.language* B
⟨*proof*⟩

end

locale *automaton-intersection-run* =
automaton-product
$automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$
$automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$
$automaton_3$ $alphabet_3$ $initial_3$ $transition_3$ $condition_3$
condition +
a: *automaton-run* $automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$ $test_1$ +
b: *automaton-run* $automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$ $test_2$ +
c: *automaton-run* $automaton_3$ $alphabet_3$ $initial_3$ $transition_3$ $condition_3$ $test_3$
for $automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$ $test_1$
and $automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$ $test_2$
and $automaton_3$ $alphabet_3$ $initial_3$ $transition_3$ $condition_3$ $test_3$
and *condition*
+

assumes *test*[*iff*]: $test_3$ (*condition* c_1 c_2) *w* (*r* ||| *s*) (*p*, *q*) $\longleftrightarrow$ $test_1$ c_1 *w r p* $\wedge$ $test_2$ c_2 *w s q*
begin

lemma *product-language*[*simp*]: *c.language* (*product A B*) = *a.language A* $\cap$ *b.language B*
⟨*proof*⟩

end

locale *automaton-sum* =
a: *automaton* $automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$ +
b: *automaton* $automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$ +
c: *automaton* $automaton_3$ $alphabet_3$ $initial_3$ $transition_3$ $condition_3$
for $automaton_1$ :: *'label set* $\Rightarrow$ *'state*$_1$ *set* $\Rightarrow$ (*'label* $\Rightarrow$ *'state*$_1$ $\Rightarrow$ *'state*$_1$ *set*) $\Rightarrow$ *'condition*$_1$ $\Rightarrow$ *'automaton*$_1$
and $alphabet_1$ $initial_1$ $transition_1$ $condition_1$
and $automaton_2$ :: *'label set* $\Rightarrow$ *'state*$_2$ *set* $\Rightarrow$ (*'label* $\Rightarrow$ *'state*$_2$ $\Rightarrow$ *'state*$_2$ *set*) $\Rightarrow$ *'condition*$_2$ $\Rightarrow$ *'automaton*$_2$
and $alphabet_2$ $initial_2$ $transition_2$ $condition_2$
and $automaton_3$ :: *'label set* $\Rightarrow$ (*'state*$_1$ + *'state*$_2$) *set* $\Rightarrow$ (*'label* $\Rightarrow$ *'state*$_1$ + *'state*$_2$ $\Rightarrow$ (*'state*$_1$ + *'state*$_2$) *set*) $\Rightarrow$ *'condition*$_3$ $\Rightarrow$ *'automaton*$_3$
and $alphabet_3$ $initial_3$ $transition_3$ $condition_3$
+
fixes *condition* :: *'condition*$_1$ $\Rightarrow$ *'condition*$_2$ $\Rightarrow$ *'condition*$_3$
begin

definition *sum* :: *'automaton*$_1$ $\Rightarrow$ *'automaton*$_2$ $\Rightarrow$ *'automaton*$_3$ **where**
sum A B $\equiv$ $automaton_3$
($alphabet_1$ *A* $\cup$ $alphabet_2$ *B*)
($initial_1$ *A* <+> $initial_2$ *B*)
(λ *a*. λ *Inl p* $\Rightarrow$ *Inl* ‘ $transition_1$ *A a p* | *Inr q* $\Rightarrow$ *Inr* ‘ $transition_2$ *B a q*)
(*condition* ($condition_1$ *A*) ($condition_2$ *B*))

lemma *sum-simps*[*simp*]:
$alphabet_3$ (*sum A B*) = $alphabet_1$ *A* $\cup$ $alphabet_2$ *B*
$initial_3$ (*sum A B*) = $initial_1$ *A* <+> $initial_2$ *B*
$transition_3$ (*sum A B*) *a* (*Inl p*) = *Inl* ‘ $transition_1$ *A a p*
$transition_3$ (*sum A B*) *a* (*Inr q*) = *Inr* ‘ $transition_2$ *B a q*
$condition_3$ (*sum A B*) = *condition* ($condition_1$ *A*) ($condition_2$ *B*)
⟨*proof*⟩

lemma *path-sum-a*:
assumes *length r* = *length w a.path A* (*w* || *r*) *p*
shows *c.path* (*sum A B*) (*w* || *map Inl r*) (*Inl p*)
⟨*proof*⟩
lemma *path-sum-b*:
assumes *length s* = *length w b.path B* (*w* || *s*) *q*
shows *c.path* (*sum A B*) (*w* || *map Inr s*) (*Inr q*)

$\langle proof \rangle$
lemma *sum-path*:
assumes $\mathit{alphabet}_1\ A = \mathit{alphabet}_2\ B$
assumes $\mathit{length}\ \mathit{rs} = \mathit{length}\ w\ \mathit{c.path}\ (\mathit{sum}\ A\ B)\ (w\ ||\ \mathit{rs})\ \mathit{pq}$
obtains
$(a)\ r\ p$ **where** $\mathit{rs} = \mathit{map}\ \mathit{Inl}\ r\ \mathit{pq} = \mathit{Inl}\ p\ \mathit{a.path}\ A\ (w\ ||\ r)\ p\ |$
$(b)\ s\ q$ **where** $\mathit{rs} = \mathit{map}\ \mathit{Inr}\ s\ \mathit{pq} = \mathit{Inr}\ q\ \mathit{b.path}\ B\ (w\ ||\ s)\ q$
$\langle proof \rangle$

lemma *run-sum-a*:
assumes $\mathit{a.run}\ A\ (w\ |||\ r)\ p$
shows $\mathit{c.run}\ (\mathit{sum}\ A\ B)\ (w\ |||\ \mathit{smap}\ \mathit{Inl}\ r)\ (\mathit{Inl}\ p)$
$\langle proof \rangle$
lemma *run-sum-b*:
assumes $\mathit{b.run}\ B\ (w\ |||\ s)\ q$
shows $\mathit{c.run}\ (\mathit{sum}\ A\ B)\ (w\ |||\ \mathit{smap}\ \mathit{Inr}\ s)\ (\mathit{Inr}\ q)$
$\langle proof \rangle$
lemma *sum-run*:
assumes $\mathit{alphabet}_1\ A = \mathit{alphabet}_2\ B$
assumes $\mathit{c.run}\ (\mathit{sum}\ A\ B)\ (w\ |||\ \mathit{rs})\ \mathit{pq}$
obtains
$(a)\ r\ p$ **where** $\mathit{rs} = \mathit{smap}\ \mathit{Inl}\ r\ \mathit{pq} = \mathit{Inl}\ p\ \mathit{a.run}\ A\ (w\ |||\ r)\ p\ |$
$(b)\ s\ q$ **where** $\mathit{rs} = \mathit{smap}\ \mathit{Inr}\ s\ \mathit{pq} = \mathit{Inr}\ q\ \mathit{b.run}\ B\ (w\ |||\ s)\ q$
$\langle proof \rangle$

lemma *sum-nodes*:
assumes $\mathit{alphabet}_1\ A = \mathit{alphabet}_2\ B$
shows $\mathit{c.nodes}\ (\mathit{sum}\ A\ B) \subseteq \mathit{a.nodes}\ A <+> \mathit{b.nodes}\ B$
$\langle proof \rangle$

lemma *sum-nodes-finite*[*intro*]:
assumes $\mathit{alphabet}_1\ A = \mathit{alphabet}_2\ B$
assumes $\mathit{finite}\ (\mathit{a.nodes}\ A)\ \mathit{finite}\ (\mathit{b.nodes}\ B)$
shows $\mathit{finite}\ (\mathit{c.nodes}\ (\mathit{sum}\ A\ B))$
$\langle proof \rangle$

end

locale *automaton-union-path* =
automaton-sum
$\mathit{automaton}_1\ \mathit{alphabet}_1\ \mathit{initial}_1\ \mathit{transition}_1\ \mathit{condition}_1$
$\mathit{automaton}_2\ \mathit{alphabet}_2\ \mathit{initial}_2\ \mathit{transition}_2\ \mathit{condition}_2$
$\mathit{automaton}_3\ \mathit{alphabet}_3\ \mathit{initial}_3\ \mathit{transition}_3\ \mathit{condition}_3$
$\mathit{condition}\ +$
a: *automaton-path* $\mathit{automaton}_1\ \mathit{alphabet}_1\ \mathit{initial}_1\ \mathit{transition}_1\ \mathit{condition}_1\ \mathit{test}_1\ +$
b: *automaton-path* $\mathit{automaton}_2\ \mathit{alphabet}_2\ \mathit{initial}_2\ \mathit{transition}_2\ \mathit{condition}_2\ \mathit{test}_2\ +$
c: *automaton-path* $\mathit{automaton}_3\ \mathit{alphabet}_3\ \mathit{initial}_3\ \mathit{transition}_3\ \mathit{condition}_3\ \mathit{test}_3$
for $\mathit{automaton}_1\ \mathit{alphabet}_1\ \mathit{initial}_1\ \mathit{transition}_1\ \mathit{condition}_1\ \mathit{test}_1$
and $\mathit{automaton}_2\ \mathit{alphabet}_2\ \mathit{initial}_2\ \mathit{transition}_2\ \mathit{condition}_2\ \mathit{test}_2$

and $automaton_3$ $alphabet_3$ $initial_3$ $transition_3$ $condition_3$ $test_3$
and *condition*
$+$
assumes $test_1[iff]$: $length\ r = length\ w \Longrightarrow test_3\ (condition\ c_1\ c_2)\ w\ (map\ Inl\ r)\ (Inl\ p) \longleftrightarrow test_1\ c_1\ w\ r\ p$
assumes $test_2[iff]$: $length\ s = length\ w \Longrightarrow test_3\ (condition\ c_1\ c_2)\ w\ (map\ Inr\ s)\ (Inr\ q) \longleftrightarrow test_2\ c_2\ w\ s\ q$
begin

lemma *sum-language*[*simp*]:
assumes $alphabet_1\ A = alphabet_2\ B$
shows $c.language\ (sum\ A\ B) = a.language\ A \cup b.language\ B$
⟨*proof*⟩

end

locale *automaton-union-run* =
automaton-sum
$automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$
$automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$
$automaton_3$ $alphabet_3$ $initial_3$ $transition_3$ $condition_3$
condition +
a: *automaton-run* $automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$ $test_1$ +
b: *automaton-run* $automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$ $test_2$ +
c: *automaton-run* $automaton_3$ $alphabet_3$ $initial_3$ $transition_3$ $condition_3$ $test_3$
for $automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$ $test_1$
and $automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$ $test_2$
and $automaton_3$ $alphabet_3$ $initial_3$ $transition_3$ $condition_3$ $test_3$
and *condition*
$+$
assumes $test_1[iff]$: $test_3\ (condition\ c_1\ c_2)\ w\ (smap\ Inl\ r)\ (Inl\ p) \longleftrightarrow test_1\ c_1\ w\ r\ p$
assumes $test_2[iff]$: $test_3\ (condition\ c_1\ c_2)\ w\ (smap\ Inr\ s)\ (Inr\ q) \longleftrightarrow test_2\ c_2\ w\ s\ q$
begin

lemma *sum-language*[*simp*]:
assumes $alphabet_1\ A = alphabet_2\ B$
shows $c.language\ (sum\ A\ B) = a.language\ A \cup b.language\ B$
⟨*proof*⟩

end

locale *automaton-product-list* =
a: *automaton* $automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$ +
b: *automaton* $automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$
for $automaton_1 :: 'label\ set \Rightarrow 'state\ set \Rightarrow ('label \Rightarrow 'state \Rightarrow 'state\ set) \Rightarrow 'condition_1 \Rightarrow 'automaton_1$
and $alphabet_1$ $initial_1$ $transition_1$ $condition_1$

and $automaton_2$ :: $'label\ set \Rightarrow 'state\ list\ set \Rightarrow ('label \Rightarrow 'state\ list \Rightarrow 'state\ list\ set) \Rightarrow 'condition_2 \Rightarrow 'automaton_2$
and $alphabet_2\ initial_2\ transition_2\ condition_2$
$+$
fixes $condition$ :: $'condition_1\ list \Rightarrow 'condition_2$
begin

definition $product$ :: $'automaton_1\ list \Rightarrow 'automaton_2$ **where**
$product\ AA \equiv automaton_2$
$(\bigcap\ (alphabet_1\ `\ set\ AA))$
$(listset\ (map\ initial_1\ AA))$
$(\lambda\ a\ ps.\ listset\ (map2\ (\lambda\ A\ p.\ transition_1\ A\ a\ p)\ AA\ ps))$
$(condition\ (map\ condition_1\ AA))$

lemma $product\text{-}simps[simp]$:
$alphabet_2\ (product\ AA) = \bigcap\ (alphabet_1\ `\ set\ AA)$
$initial_2\ (product\ AA) = listset\ (map\ initial_1\ AA)$
$transition_2\ (product\ AA)\ a\ ps = listset\ (map2\ (\lambda\ A\ p.\ transition_1\ A\ a\ p)\ AA\ ps)$
$condition_2\ (product\ AA) = condition\ (map\ condition_1\ AA)$
$\langle proof \rangle$

lemma $product\text{-}run\text{-}length$:
assumes $length\ ps = length\ AA$
assumes $b.run\ (product\ AA)\ (w\ |||\ r)\ ps$
assumes $qs \in sset\ r$
shows $length\ qs = length\ AA$
$\langle proof \rangle$
lemma $product\text{-}run\text{-}stranspose$:
assumes $length\ ps = length\ AA$
assumes $b.run\ (product\ AA)\ (w\ |||\ r)\ ps$
obtains rs **where** $r = stranspose\ rs\ length\ rs = length\ AA$
$\langle proof \rangle$

lemma $run\text{-}product$:
assumes $length\ rs = length\ AA\ length\ ps = length\ AA$
assumes $\bigwedge\ k.\ k < length\ AA \Longrightarrow a.run\ (AA\ !\ k)\ (w\ |||\ rs\ !\ k)\ (ps\ !\ k)$
shows $b.run\ (product\ AA)\ (w\ |||\ stranspose\ rs)\ ps$
$\langle proof \rangle$
lemma $product\text{-}run$:
assumes $length\ rs = length\ AA\ length\ ps = length\ AA$
assumes $b.run\ (product\ AA)\ (w\ |||\ stranspose\ rs)\ ps$
shows $k < length\ AA \Longrightarrow a.run\ (AA\ !\ k)\ (w\ |||\ rs\ !\ k)\ (ps\ !\ k)$
$\langle proof \rangle$

lemma $product\text{-}nodes$: $b.nodes\ (product\ AA) \subseteq listset\ (map\ a.nodes\ AA)$
$\langle proof \rangle$

lemma $product\text{-}nodes\text{-}finite[intro]$:

assumes *list-all* (*finite* ∘ *a.nodes*) *AA*
shows *finite* (*b.nodes* (*product AA*))
⟨*proof*⟩
lemma *product-nodes-card*:
assumes *list-all* (*finite* ∘ *a.nodes*) *AA*
shows *card* (*b.nodes* (*product AA*)) ≤ *prod-list* (*map* (*card* ∘ *a.nodes*) *AA*)
⟨*proof*⟩

end

locale *automaton-intersection-list-run* =
automaton-product-list
$automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$
$automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$
condition +
a: *automaton-run* $automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$ $test_1$ +
b: *automaton-run* $automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$ $test_2$
for $automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$ $test_1$
and $automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$ $test_2$
and *condition*
+
assumes *test*[*iff*]: *length rs* = *length cs* ⟹ *length ps* = *length cs* ⟹
$test_2$ (*condition cs*) *w* (*stranspose rs*) *ps* ⟷ *list-all* (λ (*c*, *r*, *p*). $test_1$ *c w r p*) (*cs* || *rs* || *ps*)
begin

lemma *product-language*[*simp*]: *b.language* (*product AA*) = ⋂ (*a.language* ' *set AA*)
⟨*proof*⟩

end

locale *automaton-sum-list* =
a: *automaton* $automaton_1$ $alphabet_1$ $initial_1$ $transition_1$ $condition_1$ +
b: *automaton* $automaton_2$ $alphabet_2$ $initial_2$ $transition_2$ $condition_2$
for $automaton_1$ :: *'label set* ⇒ *'state set* ⇒ (*'label* ⇒ *'state* ⇒ *'state set*) ⇒ $'condition_1$ ⇒ $'automaton_1$
and $alphabet_1$ $initial_1$ $transition_1$ $condition_1$
and $automaton_2$ :: *'label set* ⇒ (*nat* × *'state*) *set* ⇒ (*'label* ⇒ *nat* × *'state* ⇒ (*nat* × *'state*) *set*) ⇒ $'condition_2$ ⇒ $'automaton_2$
and $alphabet_2$ $initial_2$ $transition_2$ $condition_2$
+
fixes *condition* :: $'condition_1$ *list* ⇒ $'condition_2$
begin

definition *sum* :: $'automaton_1$ *list* ⇒ $'automaton_2$ **where**
sum AA ≡ $automaton_2$
(⋃ ($alphabet_1$ ' *set AA*))
(⋃ *k* < *length AA*. {*k*} × $initial_1$ (*AA* ! *k*))

$(\lambda\ a\ (k,\ p).\ \{k\} \times \mathit{transition}_1\ (AA\ !\ k)\ a\ p)$
$(\mathit{condition}\ (\mathit{map}\ \mathit{condition}_1\ AA))$

lemma *sum-simps*[*simp*]:
$\mathit{alphabet}_2\ (\mathit{sum}\ AA) = \bigcup\ (\mathit{alphabet}_1\ `\ \mathit{set}\ AA)$
$\mathit{initial}_2\ (\mathit{sum}\ AA) = (\bigcup\ k < \mathit{length}\ AA.\ \{k\} \times \mathit{initial}_1\ (AA\ !\ k))$
$\mathit{transition}_2\ (\mathit{sum}\ AA)\ a\ (k,\ p) = \{k\} \times \mathit{transition}_1\ (AA\ !\ k)\ a\ p$
$\mathit{condition}_2\ (\mathit{sum}\ AA) = \mathit{condition}\ (\mathit{map}\ \mathit{condition}_1\ AA)$
$\langle \mathit{proof} \rangle$

lemma *run-sum*:
assumes $\bigcap\ (\mathit{alphabet}_1\ `\ \mathit{set}\ AA) = \bigcup\ (\mathit{alphabet}_1\ `\ \mathit{set}\ AA)$
assumes $A \in \mathit{set}\ AA$
assumes $a.\mathit{run}\ A\ (w\ |||\ s)\ p$
obtains k **where** $k < \mathit{length}\ AA\ A = AA\ !\ k\ b.\mathit{run}\ (\mathit{sum}\ AA)\ (w\ |||\ \mathit{sconst}\ k\ |||\ s)\ (k,\ p)$
$\langle \mathit{proof} \rangle$
lemma *sum-run*:
assumes $\bigcap\ (\mathit{alphabet}_1\ `\ \mathit{set}\ AA) = \bigcup\ (\mathit{alphabet}_1\ `\ \mathit{set}\ AA)$
assumes $k < \mathit{length}\ AA$
assumes $b.\mathit{run}\ (\mathit{sum}\ AA)\ (w\ |||\ r)\ (k,\ p)$
obtains s **where** $r = \mathit{sconst}\ k\ |||\ s\ a.\mathit{run}\ (AA\ !\ k)\ (w\ |||\ s)\ p$
$\langle \mathit{proof} \rangle$

lemma *sum-nodes*:
assumes $\bigcap\ (\mathit{alphabet}_1\ `\ \mathit{set}\ AA) = \bigcup\ (\mathit{alphabet}_1\ `\ \mathit{set}\ AA)$
shows $b.\mathit{nodes}\ (\mathit{sum}\ AA) \subseteq (\bigcup\ k < \mathit{length}\ AA.\ \{k\} \times a.\mathit{nodes}\ (AA\ !\ k))$
$\langle \mathit{proof} \rangle$

lemma *sum-nodes-finite*[*intro*]:
assumes $\bigcap\ (\mathit{alphabet}_1\ `\ \mathit{set}\ AA) = \bigcup\ (\mathit{alphabet}_1\ `\ \mathit{set}\ AA)$
assumes *list-all* (*finite* $\circ$ *a.nodes*) *AA*
shows *finite* (*b.nodes* (*sum AA*))
$\langle \mathit{proof} \rangle$

end

locale *automaton-union-list-run* =
automaton-sum-list
$\mathit{automaton}_1\ \mathit{alphabet}_1\ \mathit{initial}_1\ \mathit{transition}_1\ \mathit{condition}_1$
$\mathit{automaton}_2\ \mathit{alphabet}_2\ \mathit{initial}_2\ \mathit{transition}_2\ \mathit{condition}_2$
condition +
a: *automaton-run* $\mathit{automaton}_1\ \mathit{alphabet}_1\ \mathit{initial}_1\ \mathit{transition}_1\ \mathit{condition}_1\ \mathit{test}_1$ +
b: *automaton-run* $\mathit{automaton}_2\ \mathit{alphabet}_2\ \mathit{initial}_2\ \mathit{transition}_2\ \mathit{condition}_2\ \mathit{test}_2$
for $\mathit{automaton}_1\ \mathit{alphabet}_1\ \mathit{initial}_1\ \mathit{transition}_1\ \mathit{condition}_1\ \mathit{test}_1$
and $\mathit{automaton}_2\ \mathit{alphabet}_2\ \mathit{initial}_2\ \mathit{transition}_2\ \mathit{condition}_2\ \mathit{test}_2$
and *condition*
+
assumes *test*[*iff*]: $k < \mathit{length}\ cs \Longrightarrow \mathit{test}_2\ (\mathit{condition}\ cs)\ w\ (\mathit{sconst}\ k\ |||\ r)\ (k,$

$p) \longleftrightarrow test_1\ (cs\ !\ k)\ w\ r\ p$
begin

lemma *sum-language*[*simp*]:
assumes $\bigcap\ (alphabet_1\ `\ set\ AA) = \bigcup\ (alphabet_1\ `\ set\ AA)$
shows $b.language\ (sum\ AA) = \bigcup\ (a.language\ `\ set\ AA)$
⟨*proof*⟩

end

end

13 Nondeterministic Finite Automata

theory *NFA*
imports ../*Nondeterministic*
begin

datatype $('label, 'state)\ nfa = nfa$
$(alphabet: 'label\ set)$
$(initial: 'state\ set)$
$(transition: 'label \Rightarrow 'state \Rightarrow 'state\ set)$
$(accepting: 'state\ pred)$

global-interpretation *nfa*: *automaton nfa alphabet initial transition accepting*
defines *path* = *nfa.path* **and** *run* = *nfa.run* **and** *reachable* = *nfa.reachable* **and** *nodes* = *nfa.nodes*
⟨*proof*⟩
global-interpretation *nfa*: *automaton-path nfa alphabet initial transition accepting* $\lambda\ P\ w\ r\ p.\ P\ (last\ (p\ \#\ r))$
defines *language* = *nfa.language*
⟨*proof*⟩

abbreviation *target* **where** *target* ≡ *nfa.target*
abbreviation *states* **where** *states* ≡ *nfa.states*
abbreviation *trace* **where** *trace* ≡ *nfa.trace*
abbreviation *successors* **where** *successors* ≡ *nfa.successors TYPE*('*label*)

global-interpretation *nfa*: *automaton-intersection-path*
nfa alphabet initial transition accepting $\lambda\ P\ w\ r\ p.\ P\ (last\ (p\ \#\ r))$
nfa alphabet initial transition accepting $\lambda\ P\ w\ r\ p.\ P\ (last\ (p\ \#\ r))$
nfa alphabet initial transition accepting $\lambda\ P\ w\ r\ p.\ P\ (last\ (p\ \#\ r))$
$\lambda\ c_1\ c_2\ (p, q).\ c_1\ p \wedge c_2\ q$
defines *intersect* = *nfa.product*
⟨*proof*⟩

global-interpretation *nfa*: *automaton-union-path*
nfa alphabet initial transition accepting $\lambda\ P\ w\ r\ p.\ P\ (last\ (p\ \#\ r))$

nfa alphabet initial transition accepting λ *P w r p. P* (*last* (*p* # *r*))
nfa alphabet initial transition accepting λ *P w r p. P* (*last* (*p* # *r*))
case-sum
defines *union* = *nfa.sum*
⟨*proof*⟩

end

14 Deterministic Büchi Automata

theory *DBA*
imports ../*Deterministic*
begin

datatype (*'label*, *'state*) *dba* = *dba*
(*alphabet*: *'label set*)
(*initial*: *'state*)
(*transition*: *'label* ⇒ *'state* ⇒ *'state*)
(*accepting*: *'state pred*)

global-interpretation *dba*: *automaton dba alphabet initial transition accepting*
defines *path* = *dba.path* **and** *run* = *dba.run* **and** *reachable* = *dba.reachable* **and** *nodes* = *dba.nodes*
⟨*proof*⟩
global-interpretation *dba*: *automaton-run dba alphabet initial transition accepting* λ *P w r p. infs P* (*p* ## *r*)
defines *language* = *dba.language*
⟨*proof*⟩

abbreviation *target* **where** *target* ≡ *dba.target*
abbreviation *states* **where** *states* ≡ *dba.states*
abbreviation *trace* **where** *trace* ≡ *dba.trace*

abbreviation *successors* **where** *successors* ≡ *dba.successors TYPE*(*'label*)

end

15 Deterministic Generalized Büchi Automata

theory *DGBA*
imports ../*Deterministic*
begin

datatype (*'label*, *'state*) *dgba* = *dgba*
(*alphabet*: *'label set*)
(*initial*: *'state*)
(*transition*: *'label* ⇒ *'state* ⇒ *'state*)

(*accepting*: *'state pred gen*)

global-interpretation *dgba*: *automaton dgba alphabet initial transition accepting*
defines *path* = *dgba.path* **and** *run* = *dgba.run* **and** *reachable* = *dgba.reachable* **and** *nodes* = *dgba.nodes*
⟨*proof*⟩
global-interpretation *dgba*: *automaton-run dgba alphabet initial transition accepting* λ *P w r p. gen infs P* (*p ## r*)
defines *language* = *dgba.language*
⟨*proof*⟩

abbreviation *target* **where** *target* ≡ *dgba.target*
abbreviation *states* **where** *states* ≡ *dgba.states*
abbreviation *trace* **where** *trace* ≡ *dgba.trace*
abbreviation *successors* **where** *successors* ≡ *dgba.successors TYPE*(*'label*)

end

16 Deterministic Büchi Automata Combinations

theory *DBA-Combine*
imports *DBA DGBA*
begin

global-interpretation *degeneralization*: *automaton-degeneralization-run*
dgba dgba.alphabet dgba.initial dgba.transition dgba.accepting λ *P w r p. gen infs P* (*p ## r*)
dba dba.alphabet dba.initial dba.transition dba.accepting λ *P w r p. infs P* (*p ## r*)
fst id
defines *degeneralize* = *degeneralization.degeneralize*
⟨*proof*⟩

lemmas *degeneralize-language*[*simp*] = *degeneralization.degeneralize-language*[*folded DBA.language-def*]
lemmas *degeneralize-nodes-finite*[*iff*] = *degeneralization.degeneralize-nodes-finite*[*folded DBA.nodes-def*]
lemmas *degeneralize-nodes-card* = *degeneralization.degeneralize-nodes-card*[*folded DBA.nodes-def*]

global-interpretation *intersection*: *automaton-intersection-run*
dba.dba dba.alphabet dba.initial dba.transition dba.accepting λ *P w r p. infs P* (*p ## r*)
dba.dba dba.alphabet dba.initial dba.transition dba.accepting λ *P w r p. infs P* (*p ## r*)
dgba.dgba dgba.alphabet dgba.initial dgba.transition dgba.accepting λ *P w r p. gen infs P* (*p ## r*)
λ c_1 c_2. [$c_1 \circ$ *fst*, $c_2 \circ$ *snd*]
defines *intersect'* = *intersection.product*

⟨*proof*⟩

lemmas *intersect′-language*[*simp*] = *intersection.product-language*[*folded DGBA.language-def*]
lemmas *intersect′-nodes-finite* = *intersection.product-nodes-finite*[*folded DGBA.nodes-def*]
lemmas *intersect′-nodes-card* = *intersection.product-nodes-card*[*folded DGBA.nodes-def*]

global-interpretation *union*: *automaton-union-run*
dba.dba dba.alphabet dba.initial dba.transition dba.accepting λ *P w r p. infs P* (*p ## r*)
dba.dba dba.alphabet dba.initial dba.transition dba.accepting λ *P w r p. infs P* (*p ## r*)
dba.dba dba.alphabet dba.initial dba.transition dba.accepting λ *P w r p. infs P* (*p ## r*)
$\lambda\ c_1\ c_2\ pq.\ (c_1 \circ fst)\ pq \lor (c_2 \circ snd)\ pq$
defines *union* = *union.product*
⟨*proof*⟩

lemmas *union-language* = *union.product-language*
lemmas *union-nodes-finite* = *union.product-nodes-finite*
lemmas *union-nodes-card* = *union.product-nodes-card*

global-interpretation *intersection-list*: *automaton-intersection-list-run*
dba.dba dba.alphabet dba.initial dba.transition dba.accepting λ *P w r p. infs P* (*p ## r*)
dgba.dgba dgba.alphabet dgba.initial dgba.transition dgba.accepting λ *P w r p. gen infs P* (*p ## r*)
λ *cs. map* (λ *k pp.* (*cs* ! *k*) (*pp* ! *k*)) [*0* ..< *length cs*]
defines *intersect-list′* = *intersection-list.product*
⟨*proof*⟩

lemmas *intersect-list′-language*[*simp*] = *intersection-list.product-language*[*folded DGBA.language-def*]
lemmas *intersect-list′-nodes-finite* = *intersection-list.product-nodes-finite*[*folded DGBA.nodes-def*]
lemmas *intersect-list′-nodes-card* = *intersection-list.product-nodes-card*[*folded DGBA.nodes-def*]

global-interpretation *union-list*: *automaton-union-list-run*
dba.dba dba.alphabet dba.initial dba.transition dba.accepting λ *P w r p. infs P* (*p ## r*)
dba.dba dba.alphabet dba.initial dba.transition dba.accepting λ *P w r p. infs P* (*p ## r*)
λ *cs pp.* $\exists$ *k* < *length cs.* (*cs* ! *k*) (*pp* ! *k*)
defines *union-list* = *union-list.product*
⟨*proof*⟩

lemmas *union-list-language* = *union-list.product-language*
lemmas *union-list-nodes-finite* = *union-list.product-nodes-finite*
lemmas *union-list-nodes-card* = *union-list.product-nodes-card*

abbreviation *intersect* **where** *intersect A B* ≡ *degeneralize* (*intersect′ A B*)

lemma *intersect-language*[*simp*]: *DBA.language* (*intersect A B*) = *DBA.language A* ∩ *DBA.language B*
⟨*proof*⟩
lemma *intersect-nodes-finite*:
assumes *finite* (*DBA.nodes A*) *finite* (*DBA.nodes B*)
shows *finite* (*DBA.nodes* (*intersect A B*))
⟨*proof*⟩
lemma *intersect-nodes-card*:
assumes *finite* (*DBA.nodes A*) *finite* (*DBA.nodes B*)
shows *card* (*DBA.nodes* (*intersect A B*)) ≤ *2* ∗ *card* (*DBA.nodes A*) ∗ *card* (*DBA.nodes B*)
⟨*proof*⟩

abbreviation *intersect-list* **where** *intersect-list AA* ≡ *degeneralize* (*intersect-list′ AA*)

lemma *intersect-list-language*[*simp*]: *DBA.language* (*intersect-list AA*) = ⋂ (*DBA.language* ‘ *set AA*)
⟨*proof*⟩
lemma *intersect-list-nodes-finite*:
assumes *list-all* (*finite* ∘ *DBA.nodes*) *AA*
shows *finite* (*DBA.nodes* (*intersect-list AA*))
⟨*proof*⟩
lemma *intersect-list-nodes-card*:
assumes *list-all* (*finite* ∘ *DBA.nodes*) *AA*
shows *card* (*DBA.nodes* (*intersect-list AA*)) ≤ *max 1* (*length AA*) ∗ *prod-list* (*map* (*card* ∘ *DBA.nodes*) *AA*)
⟨*proof*⟩

end

17 Deterministic Büchi Transition Automata

theory *DBTA*
imports ../*Deterministic*
begin

datatype (*′label*, *′state*) *dbta* = *dbta*
(*alphabet*: *′label set*)
(*initial*: *′state*)
(*transition*: *′label* ⇒ *′state* ⇒ *′state*)
(*accepting*: (*′state* × *′label* × *′state*) *pred*)

global-interpretation *dbta*: *automaton dbta alphabet initial transition accepting*

defines *path* = *dbta.path* **and** *run* = *dbta.run* **and** *reachable* = *dbta.reachable* **and** *nodes* = *dbta.nodes*
⟨*proof*⟩
global-interpretation *dbta*: *automaton-run dbta alphabet initial transition accepting*
λ *P w r p. infs P* (*p* ## *r* ||| *w* ||| *r*)
defines *language* = *dbta.language*
⟨*proof*⟩

abbreviation *target* **where** *target* ≡ *dbta.target*
abbreviation *states* **where** *states* ≡ *dbta.states*
abbreviation *trace* **where** *trace* ≡ *dbta.trace*
abbreviation *successors* **where** *successors* ≡ *dbta.successors TYPE*('*label*)

end

18 Deterministic Generalized Büchi Transition Automata

theory *DGBTA*
imports *../Deterministic*
begin

datatype ('*label*, '*state*) *dgbta* = *dgbta*
(*alphabet*: '*label set*)
(*initial*: '*state*)
(*transition*: '*label* ⇒ '*state* ⇒ '*state*)
(*accepting*: ('*state* × '*label* × '*state*) *pred gen*)

global-interpretation *dgbta*: *automaton dgbta alphabet initial transition accepting*
defines *path* = *dgbta.path* **and** *run* = *dgbta.run* **and** *reachable* = *dgbta.reachable* **and** *nodes* = *dgbta.nodes*
⟨*proof*⟩
global-interpretation *dgbta*: *automaton-run dgbta alphabet initial transition accepting*
λ *P w r p. gen infs P* (*p* ## *r* ||| *w* ||| *r*)
defines *language* = *dgbta.language*
⟨*proof*⟩

abbreviation *target* **where** *target* ≡ *dgbta.target*
abbreviation *states* **where** *states* ≡ *dgbta.states*
abbreviation *trace* **where** *trace* ≡ *dgbta.trace*
abbreviation *successors* **where** *successors* ≡ *dgbta.successors TYPE*('*label*)

end

19 Deterministic Büchi Transition Automata Combinations

theory *DBTA-Combine*
imports *DBTA DGBTA*
begin

global-interpretation *degeneralization*: *automaton-degeneralization-run*
dgbta dgbta.alphabet dgbta.initial dgbta.transition dgbta.accepting λ *P w r p. gen infs P* (*p* ## *r* ||| *w* ||| *r*)
dbta dbta.alphabet dbta.initial dbta.transition dbta.accepting λ *P w r p. infs P* (*p* ## *r* ||| *w* ||| *r*)
id λ ((*p*, *k*), *a*, (*q*, *l*)). ((*p*, *a*, *q*), *k*)
defines *degeneralize* = *degeneralization.degeneralize*
⟨*proof*⟩

lemmas *degeneralize-language*[*simp*] = *degeneralization.degeneralize-language*[*folded DBTA.language-def*]
lemmas *degeneralize-nodes-finite*[*iff*] = *degeneralization.degeneralize-nodes-finite*[*folded DBTA.nodes-def*]
lemmas *degeneralize-nodes-card* = *degeneralization.degeneralize-nodes-card*[*folded DBTA.nodes-def*]

global-interpretation *intersection*: *automaton-intersection-run*
dbta.dbta dbta.alphabet dbta.initial dbta.transition dbta.accepting λ *P w r p. infs P* (*p* ## *r* ||| *w* ||| *r*)
dbta.dbta dbta.alphabet dbta.initial dbta.transition dbta.accepting λ *P w r p. infs P* (*p* ## *r* ||| *w* ||| *r*)
dgbta.dgbta dgbta.alphabet dgbta.initial dgbta.transition dgbta.accepting λ *P w r p. gen infs P* (*p* ## *r* ||| *w* ||| *r*)
λ c_1 c_2. [c_1 $\circ$ (λ ((p_1, p_2), *a*, (q_1, q_2)). (p_1, *a*, q_1)), c_2 $\circ$ (λ ((p_1, p_2), *a*, (q_1, q_2)). (p_2, *a*, q_2))]
defines *intersect′* = *intersection.product*
⟨*proof*⟩

lemmas *intersect′-language*[*simp*] = *intersection.product-language*[*folded DGBTA.language-def*]
lemmas *intersect′-nodes-finite* = *intersection.product-nodes-finite*[*folded DGBTA.nodes-def*]
lemmas *intersect′-nodes-card* = *intersection.product-nodes-card*[*folded DGBTA.nodes-def*]

global-interpretation *union*: *automaton-union-run*
dbta.dbta dbta.alphabet dbta.initial dbta.transition dbta.accepting λ *P w r p. infs P* (*p* ## *r* ||| *w* ||| *r*)
dbta.dbta dbta.alphabet dbta.initial dbta.transition dbta.accepting λ *P w r p. infs P* (*p* ## *r* ||| *w* ||| *r*)
dbta.dbta dbta.alphabet dbta.initial dbta.transition dbta.accepting λ *P w r p. infs P* (*p* ## *r* ||| *w* ||| *r*)
λ c_1 c_2 *pq*. (c_1 $\circ$ (λ ((p_1, p_2), *a*, (q_1, q_2)). (p_1, *a*, q_1))) *pq* $\vee$ (c_2 $\circ$ (λ ((p_1, p_2),

$a,\ (q_1,\ q_2)).\ (p_2,\ a,\ q_2)))\ pq$
defines *union* = *union.product*
⟨*proof*⟩

lemmas *union-language* = *union.product-language*
lemmas *union-nodes-finite* = *union.product-nodes-finite*
lemmas *union-nodes-card* = *union.product-nodes-card*

abbreviation *intersect* **where** *intersect A B* ≡ *degeneralize* (*intersect′ A B*)

lemma *intersect-language*[*simp*]: *DBTA.language* (*intersect A B*) = *DBTA.language A* ∩ *DBTA.language B*
⟨*proof*⟩
lemma *intersect-nodes-finite*:
assumes *finite* (*DBTA.nodes A*) *finite* (*DBTA.nodes B*)
shows *finite* (*DBTA.nodes* (*intersect A B*))
⟨*proof*⟩
lemma *intersect-nodes-card*:
assumes *finite* (*DBTA.nodes A*) *finite* (*DBTA.nodes B*)
shows *card* (*DBTA.nodes* (*intersect A B*)) ≤ *2* ∗ *card* (*DBTA.nodes A*) ∗ *card* (*DBTA.nodes B*)
⟨*proof*⟩

end

20 Deterministic Co-Büchi Automata

theory *DCA*
imports ../*Deterministic*
begin

datatype (′*label*, ′*state*) *dca* = *dca*
(*alphabet*: ′*label set*)
(*initial*: ′*state*)
(*transition*: ′*label* ⇒ ′*state* ⇒ ′*state*)
(*rejecting*: ′*state* ⇒ *bool*)

global-interpretation *dca*: *automaton dca alphabet initial transition rejecting*
defines *path* = *dca.path* **and** *run* = *dca.run* **and** *reachable* = *dca.reachable* **and** *nodes* = *dca.nodes*
⟨*proof*⟩
global-interpretation *dca*: *automaton-run dca alphabet initial transition rejecting* λ *P w r p. fins P* (*p* ## *r*)
defines *language* = *dca.language*
⟨*proof*⟩

abbreviation *target* **where** *target* ≡ *dca.target*
abbreviation *states* **where** *states* ≡ *dca.states*
abbreviation *trace* **where** *trace* ≡ *dca.trace*

abbreviation *successors* **where** *successors* ≡ *dca.successors TYPE*('*label*)

end

21 Deterministic Co-Generalized Co-Büchi Automata

theory *DGCA*
imports ../*Deterministic*
begin

datatype ('*label*, '*state*) *dgca* = *dgca*
(*alphabet*: '*label set*)
(*initial*: '*state*)
(*transition*: '*label* ⇒ '*state* ⇒ '*state*)
(*rejecting*: '*state pred gen*)

global-interpretation *dgca*: *automaton dgca alphabet initial transition rejecting*
defines *path* = *dgca.path* **and** *run* = *dgca.run* **and** *reachable* = *dgca.reachable* **and** *nodes* = *dgca.nodes*
⟨*proof*⟩
global-interpretation *dgca*: *automaton-run dgca alphabet initial transition rejecting* λ *P w r p. cogen fins P* (*p* ## *r*)
defines *language* = *dgca.language*
⟨*proof*⟩

abbreviation *target* **where** *target* ≡ *dgca.target*
abbreviation *states* **where** *states* ≡ *dgca.states*
abbreviation *trace* **where** *trace* ≡ *dgca.trace*
abbreviation *successors* **where** *successors* ≡ *dgca.successors TYPE*('*label*)

end

22 Deterministic Co-Büchi Automata Combinations

theory *DCA-Combine*
imports *DCA DGCA*
begin

global-interpretation *degeneralization*: *automaton-degeneralization-run*
dgca dgca.alphabet dgca.initial dgca.transition dgca.rejecting λ *P w r p. cogen fins P* (*p* ## *r*)
dca dca.alphabet dca.initial dca.transition dca.rejecting λ *P w r p. fins P* (*p* ## *r*)
fst id
defines *degeneralize* = *degeneralization.degeneralize*
⟨*proof*⟩

lemmas *degeneralize-language*[*simp*] = *degeneralization.degeneralize-language*[*folded DCA.language-def*]
lemmas *degeneralize-nodes-finite*[*iff*] = *degeneralization.degeneralize-nodes-finite*[*folded DCA.nodes-def*]
lemmas *degeneralize-nodes-card* = *degeneralization.degeneralize-nodes-card*[*folded DCA.nodes-def*]

global-interpretation *intersection*: *automaton-intersection-run*
dca.dca dca.alphabet dca.initial dca.transition dca.rejecting λ *P w r p. fins P* (*p ## r*)
dca.dca dca.alphabet dca.initial dca.transition dca.rejecting λ *P w r p. fins P* (*p ## r*)
dca.dca dca.alphabet dca.initial dca.transition dca.rejecting λ *P w r p. fins P* (*p ## r*)
λ c_1 c_2 *pq*. (c_1 $\circ$ *fst*) *pq* $\vee$ (c_2 $\circ$ *snd*) *pq*
defines *intersect* = *intersection.product*
⟨*proof*⟩

lemmas *intersect-language* = *intersection.product-language*
lemmas *intersect-nodes-finite* = *intersection.product-nodes-finite*
lemmas *intersect-nodes-card* = *intersection.product-nodes-card*

global-interpretation *union*: *automaton-union-run*
dca.dca dca.alphabet dca.initial dca.transition dca.rejecting λ *P w r p. fins P* (*p ## r*)
dca.dca dca.alphabet dca.initial dca.transition dca.rejecting λ *P w r p. fins P* (*p ## r*)
dgca.dgca dgca.alphabet dgca.initial dgca.transition dgca.rejecting λ *P w r p. cogen fins P* (*p ## r*)
λ c_1 c_2. [c_1 $\circ$ *fst*, c_2 $\circ$ *snd*]
defines *union'* = *union.product*
⟨*proof*⟩

lemmas *union'-language*[*simp*] = *union.product-language*[*folded DGCA.language-def*]
lemmas *union'-nodes-finite* = *union.product-nodes-finite*[*folded DGCA.nodes-def*]
lemmas *union'-nodes-card* = *union.product-nodes-card*[*folded DGCA.nodes-def*]

global-interpretation *intersection-list*: *automaton-intersection-list-run*
dca.dca dca.alphabet dca.initial dca.transition dca.rejecting λ *P w r p. fins P* (*p ## r*)
dca.dca dca.alphabet dca.initial dca.transition dca.rejecting λ *P w r p. fins P* (*p ## r*)
λ *cs pp*. $\exists$ *k* < *length cs*. (*cs* ! *k*) (*pp* ! *k*)
defines *intersect-list* = *intersection-list.product*
⟨*proof*⟩

lemmas *intersect-list-language* = *intersection-list.product-language*
lemmas *intersect-list-nodes-finite* = *intersection-list.product-nodes-finite*
lemmas *intersect-list-nodes-card* = *intersection-list.product-nodes-card*

global-interpretation *union-list*: *automaton-union-list-run*
dca.dca dca.alphabet dca.initial dca.transition dca.rejecting λ *P w r p. fins P* (*p* ## *r*)
dgca.dgca dgca.alphabet dgca.initial dgca.transition dgca.rejecting λ *P w r p. cogen fins P* (*p* ## *r*)
λ *cs. map* (λ *k pp.* (*cs* ! *k*) (*pp* ! *k*)) [*0* ..< *length cs*]
defines *union-list′* = *union-list.product*
⟨*proof*⟩

lemmas *union-list′-language*[*simp*] = *union-list.product-language*[*folded DGCA.language-def*]
lemmas *union-list′-nodes-finite* = *union-list.product-nodes-finite*[*folded DGCA.nodes-def*]
lemmas *union-list′-nodes-card* = *union-list.product-nodes-card*[*folded DGCA.nodes-def*]

abbreviation *union* **where** *union A B* ≡ *degeneralize* (*union′ A B*)

lemma *union-language*[*simp*]:
assumes *dca.alphabet A* = *dca.alphabet B*
shows *DCA.language* (*union A B*) = *DCA.language A* ∪ *DCA.language B*
⟨*proof*⟩
lemma *union-nodes-finite*:
assumes *finite* (*DCA.nodes A*) *finite* (*DCA.nodes B*)
shows *finite* (*DCA.nodes* (*union A B*))
⟨*proof*⟩
lemma *union-nodes-card*:
assumes *finite* (*DCA.nodes A*) *finite* (*DCA.nodes B*)
shows *card* (*DCA.nodes* (*union A B*)) ≤ *2* ∗ *card* (*DCA.nodes A*) ∗ *card* (*DCA.nodes B*)
⟨*proof*⟩

abbreviation *union-list* **where** *union-list AA* ≡ *degeneralize* (*union-list′ AA*)

lemma *union-list-language*[*simp*]:
assumes $\bigcap$ (*dca.alphabet* ‘ *set AA*) = $\bigcup$ (*dca.alphabet* ‘ *set AA*)
shows *DCA.language* (*union-list AA*) = $\bigcup$ (*DCA.language* ‘ *set AA*)
⟨*proof*⟩
lemma *union-list-nodes-finite*:
assumes *list-all* (*finite* ∘ *DCA.nodes*) *AA*
shows *finite* (*DCA.nodes* (*union-list AA*))
⟨*proof*⟩
lemma *union-list-nodes-card*:
assumes *list-all* (*finite* ∘ *DCA.nodes*) *AA*
shows *card* (*DCA.nodes* (*union-list AA*)) ≤ *max 1* (*length AA*) ∗ *prod-list* (*map* (*card* ∘ *DCA.nodes*) *AA*)
⟨*proof*⟩

end

23 Deterministic Rabin Automata

theory *DRA*
imports *../Deterministic*
begin

datatype ($'label$, $'state$) *dra* = *dra*
(*alphabet*: $'label$ *set*)
(*initial*: $'state$)
(*transition*: $'label \Rightarrow 'state \Rightarrow 'state$)
(*condition*: $'state$ *rabin gen*)

global-interpretation *dra*: *automaton dra alphabet initial transition condition*
defines *path* = *dra.path* **and** *run* = *dra.run* **and** *reachable* = *dra.reachable* **and** *nodes* = *dra.nodes*
⟨*proof*⟩
global-interpretation *dra*: *automaton-run dra alphabet initial transition condition* λ *P w r p. cogen rabin P* (*p* ## *r*)
defines *language* = *dra.language*
⟨*proof*⟩

abbreviation *target* **where** *target* ≡ *dra.target*
abbreviation *states* **where** *states* ≡ *dra.states*
abbreviation *trace* **where** *trace* ≡ *dra.trace*
abbreviation *successors* **where** *successors* ≡ *dra.successors TYPE*($'label$)

end

24 Deterministic Rabin Automata Combinations

theory *DRA-Combine*
imports *DRA ../DBA/DBA ../DCA/DCA*
begin

global-interpretation *intersection-bc*: *automaton-intersection-run*
dba.dba dba.alphabet dba.initial dba.transition dba.accepting λ *P w r p. infs P* (*p* ## *r*)
dca.dca dca.alphabet dca.initial dca.transition dca.rejecting λ *P w r p. fins P* (*p* ## *r*)
dra.dra dra.alphabet dra.initial dra.transition dra.condition λ *P w r p. cogen rabin P* (*p* ## *r*)
λ c_1 c_2. [($c_1 \circ$ *fst*, $c_2 \circ$ *snd*)]
defines *intersect-bc* = *intersection-bc.product*
⟨*proof*⟩

lemmas *intersect-bc-language*[*simp*] = *intersection-bc.product-language*[*folded DCA.language-def DRA.language-def*]
lemmas *intersect-bc-nodes-finite* = *intersection-bc.product-nodes-finite*[*folded DCA.nodes-def*

DRA.nodes-def]
lemmas *intersect-bc-nodes-card* = *intersection-bc.product-nodes-card*[*folded DCA.nodes-def DRA.nodes-def*]

global-interpretation *union-list*: *automaton-union-list-run*
dra.dra dra.alphabet dra.initial dra.transition dra.condition λ *P w r p. cogen rabin P* (*p* ## *r*)
dra.dra dra.alphabet dra.initial dra.transition dra.condition λ *P w r p. cogen rabin P* (*p* ## *r*)
λ *cs. do* { *k* ← [*0* ..< *length cs*]; (*f*, *g*) ← *cs* ! *k*; [(λ *pp. f* (*pp* ! *k*), λ *pp. g* (*pp* ! *k*))] }
defines *union-list* = *union-list.product*
⟨*proof*⟩

lemmas *union-list-language* = *union-list.product-language*
lemmas *union-list-nodes-finite* = *union-list.product-nodes-finite*
lemmas *union-list-nodes-card* = *union-list.product-nodes-card*

end

25 Relations and Refinement

theory *Refine*
imports
Automatic-Refinement.Automatic-Refinement

Refine-Monadic.Refine-Foreach
Sequence-LTL
Maps
begin

25.1 Predicate to Set Conversion Setup

lemma *right-unique-pred-set-conv*[*pred-set-conv*]: *right-unique* = *single-valuedp*
⟨*proof*⟩
lemma *bi-unique-pred-set-conv*[*pred-set-conv*]: *bi-unique* (λ *x y*. (*x*, *y*) ∈ *R*) ⟷ *bijective R*
⟨*proof*⟩

useful for unfolding equality constants in theorems about predicates

lemma *pred-Id*: *HOL.eq* = (λ *x y*. (*x*, *y*) ∈ *Id*) ⟨*proof*⟩
lemma *pred-bool-Id*: *HOL.eq* = (λ *x y*. (*x*, *y*) ∈ (*Id* :: *bool rel*)) ⟨*proof*⟩
lemma *pred-nat-Id*: *HOL.eq* = (λ *x y*. (*x*, *y*) ∈ (*Id* :: *nat rel*)) ⟨*proof*⟩
lemma *pred-set-Id*: *HOL.eq* = (λ *x y*. (*x*, *y*) ∈ (*Id* :: *'a set rel*)) ⟨*proof*⟩
lemma *pred-list-Id*: *HOL.eq* = (λ *x y*. (*x*, *y*) ∈ (*Id* :: *'a list rel*)) ⟨*proof*⟩
lemma *pred-stream-Id*: *HOL.eq* = (λ *x y*. (*x*, *y*) ∈ (*Id* :: *'a stream rel*)) ⟨*proof*⟩

lemma *eq-onp-Id-on-eq*[*pred-set-conv*]: *eq-onp* (λ *a*. $a \in A$) = (λ *x y*. $(x, y) \in$ *Id-on* A)
⟨*proof*⟩
lemma *rel-fun-fun-rel-eq*[*pred-set-conv*]:
rel-fun (λ *x y*. $(x, y) \in A$) (λ *x y*. $(x, y) \in B$) = (λ *f g*. $(f, g) \in A \to B$)
⟨*proof*⟩
lemma *rel-prod-prod-rel-eq*[*pred-set-conv*]:
rel-prod (λ *x y*. $(x, y) \in A$) (λ *x y*. $(x, y) \in B$) = (λ *f g*. $(f, g) \in A \times_r B$)
⟨*proof*⟩
lemma *rel-sum-sum-rel-eq*[*pred-set-conv*]:
rel-sum (λ *x y*. $(x, y) \in A$) (λ *x y*. $(x, y) \in B$) = (λ *f g*. $(f, g) \in \langle A, B\rangle$ *sum-rel*)
⟨*proof*⟩
lemma *rel-set-set-rel-eq*[*pred-set-conv*]:
rel-set (λ *x y*. $(x, y) \in A$) = (λ *f g*. $(f, g) \in \langle A\rangle$ *set-rel*)
⟨*proof*⟩
lemma *rel-option-option-rel-eq*[*pred-set-conv*]:
rel-option (λ *x y*. $(x, y) \in A$) = (λ *f g*. $(f, g) \in \langle A\rangle$ *option-rel*)
⟨*proof*⟩

thm *image-transfer image-transfer*[*to-set*]
thm *fun-upd-transfer fun-upd-transfer*[*to-set*]

25.2 Relation Composition

lemma *relcomp-trans-1*[*trans*]:
assumes $(f, g) \in A_1$
assumes $(g, h) \in A_2$
shows $(f, h) \in A_1\ O\ A_2$
⟨*proof*⟩
lemma *relcomp-trans-2*[*trans*]:
assumes $(f, g) \in A_1 \to B_1$
assumes $(g, h) \in A_2 \to B_2$
shows $(f, h) \in A_1\ O\ A_2 \to B_1\ O\ B_2$
⟨*proof*⟩
lemma *relcomp-trans-3*[*trans*]:
assumes $(f, g) \in A_1 \to B_1 \to C_1$
assumes $(g, h) \in A_2 \to B_2 \to C_2$
shows $(f, h) \in A_1\ O\ A_2 \to B_1\ O\ B_2 \to C_1\ O\ C_2$
⟨*proof*⟩
lemma *relcomp-trans-4*[*trans*]:
assumes $(f, g) \in A_1 \to B_1 \to C_1 \to D_1$
assumes $(g, h) \in A_2 \to B_2 \to C_2 \to D_2$
shows $(f, h) \in A_1\ O\ A_2 \to B_1\ O\ B_2 \to C_1\ O\ C_2 \to D_1\ O\ D_2$
⟨*proof*⟩
lemma *relcomp-trans-5*[*trans*]:
assumes $(f, g) \in A_1 \to B_1 \to C_1 \to D_1 \to E_1$
assumes $(g, h) \in A_2 \to B_2 \to C_2 \to D_2 \to E_2$
shows $(f, h) \in A_1\ O\ A_2 \to B_1\ O\ B_2 \to C_1\ O\ C_2 \to D_1\ O\ D_2 \to E_1\ O\ E_2$

$\langle proof \rangle$

25.3 Relation Basics

lemma *inv-fun-rel-eq*[*simp*]: $(A \rightarrow B)^{-1} = A^{-1} \rightarrow B^{-1}$
$\langle proof \rangle$
lemma *inv-option-rel-eq*[*simp*]: $(\langle K \rangle option\text{-}rel)^{-1} = \langle K^{-1} \rangle option\text{-}rel$
$\langle proof \rangle$
lemma *inv-prod-rel-eq*[*simp*]: $(P \times_r Q)^{-1} = P^{-1} \times_r Q^{-1}$
$\langle proof \rangle$
lemma *inv-sum-rel-eq*[*simp*]: $(\langle P, Q \rangle sum\text{-}rel)^{-1} = \langle P^{-1}, Q^{-1} \rangle sum\text{-}rel$
$\langle proof \rangle$
lemma *set-rel-converse*[*simp*]: $(\langle A \rangle\ set\text{-}rel)^{-1} = \langle A^{-1} \rangle\ set\text{-}rel$ $\langle proof \rangle$

lemma *build-rel-domain*[*simp*]: $Domain\ (br\ \alpha\ I) = Collect\ I$ $\langle proof \rangle$
lemma *build-rel-range*[*simp*]: $Range\ (br\ \alpha\ I) = \alpha\ `\ Collect\ I$ $\langle proof \rangle$
lemma *build-rel-image*[*simp*]: $br\ \alpha\ I\ ``\ A = \alpha\ `\ (A \cap Collect\ I)$ $\langle proof \rangle$

lemma *prod-rel-domain*[*simp*]: $Domain\ (A \times_r B) = Domain\ A \times Domain\ B$
$\langle proof \rangle$
lemma *prod-rel-range*[*simp*]: $Range\ (A \times_r B) = Range\ A \times Range\ B$ $\langle proof \rangle$

lemma *member-Id-on*[*iff*]: $(x, y) \in Id\text{-}on\ A \longleftrightarrow x = y \wedge y \in A$ $\langle proof \rangle$
lemma *bijective-Id-on*[*intro*!, *simp*]: *bijective* $(Id\text{-}on\ A)$ $\langle proof \rangle$
lemma *relcomp-Id-on*[*simp*]: $Id\text{-}on\ A\ O\ Id\text{-}on\ B = Id\text{-}on\ (A \cap B)$ $\langle proof \rangle$

lemma *prod-rel-Id-on*[*simp*]: $Id\text{-}on\ A \times_r Id\text{-}on\ B = Id\text{-}on\ (A \times B)$ $\langle proof \rangle$
lemma *set-rel-Id-on*[*simp*]: $\langle Id\text{-}on\ S \rangle\ set\text{-}rel = Id\text{-}on\ (Pow\ S)$ $\langle proof \rangle$

25.4 Parametricity

lemmas *basic-param*[*param*] =
option.rel-transfer[*unfolded pred-bool-Id*, *to-set*]
All-transfer[*unfolded pred-bool-Id*, *to-set*]
Ex-transfer[*unfolded pred-bool-Id*, *to-set*]
Union-transfer[*to-set*]
image-transfer[*to-set*]
Image-parametric[*to-set*]

lemma *Sigma-param*[*param*]: $(Sigma, Sigma) \in \langle A \rangle\ set\text{-}rel \rightarrow (A \rightarrow \langle B \rangle\ set\text{-}rel) \rightarrow \langle A \times_r B \rangle\ set\text{-}rel$
$\langle proof \rangle$

lemma *set-filter-param*[*param*]:
$(Set.filter, Set.filter) \in (A \rightarrow bool\text{-}rel) \rightarrow \langle A \rangle\ set\text{-}rel \rightarrow \langle A \rangle\ set\text{-}rel$
$\langle proof \rangle$
lemma *is-singleton-param*[*param*]:
assumes *bijective A*
shows $(is\text{-}singleton, is\text{-}singleton) \in \langle A \rangle\ set\text{-}rel \rightarrow bool\text{-}rel$
$\langle proof \rangle$

lemma *the-elem-param*[*param*]:
assumes *is-singleton S is-singleton T*
assumes $(S,\ T) \in \langle A\rangle$ *set-rel*
shows (*the-elem S*, *the-elem T*) $\in A$
⟨*proof*⟩

25.5 Lists

lemma *list-all2-list-rel-conv*[*pred-set-conv*]:
list-all2 $(\lambda\ x\ y.\ (x,\ y) \in R) = (\lambda\ x\ y.\ (x,\ y) \in \langle R\rangle$ *list-rel*)
⟨*proof*⟩

lemmas *list-rel-single-valued*[*iff*] = *list-rel-sv-iff*

lemmas *list-rel-simps*[*simp*] =
list.rel-eq-onp[*to-set*]
list.rel-conversep[*to-set*, *symmetric*]
list.rel-compp[*to-set*]

lemmas *list-rel-param*[*param*] =
list.set-transfer[*to-set*]
list.pred-transfer[*unfolded pred-bool-Id*, *to-set*, *folded pred-list-listsp*]
list.rel-transfer[*unfolded pred-bool-Id*, *to-set*]

lemmas *null-param*[*param*] = *null-transfer*[*unfolded pred-bool-Id*, *to-set*]

thm *param-set list.set-transfer*[*to-set*]

lemmas *scan-param*[*param*] = *scan.transfer*[*to-set*]
lemma *bind-param*[*param*]: (*List.bind*, *List.bind*) $\in \langle A\rangle$ *list-rel* $\rightarrow (A \rightarrow \langle B\rangle$ *list-rel*) $\rightarrow \langle B\rangle$ *list-rel*
⟨*proof*⟩

lemma *set-id-param*[*param*]: (*set*, *id*) $\in \langle A\rangle$ *list-set-rel* $\rightarrow \langle A\rangle$ *set-rel*
⟨*proof*⟩

25.6 Streams

definition *stream-rel* :: $('a \times 'b)$ *set* $\Rightarrow$ $('a$ *stream* $\times\ 'b$ *stream*) *set* **where**
[*to-relAPP*]: *stream-rel* $R \equiv \{(x,\ y).$ *stream-all2* $(\lambda\ x\ y.\ (x,\ y) \in R)\ x\ y\}$

lemma *stream-all2-stream-rel-conv*[*pred-set-conv*]:
stream-all2 $(\lambda\ x\ y.\ (x,\ y) \in R) = (\lambda\ x\ y.\ (x,\ y) \in \langle R\rangle$ *stream-rel*)
⟨*proof*⟩

lemmas *stream-rel-coinduct'*[*case-names stream-rel*, *coinduct set*: *stream-rel*] =
stream-rel-coinduct[*to-set*]

lemmas *stream-rel-intros* = *stream.rel-intros*[*to-set*]

lemmas *stream-rel-cases* = *stream.rel-cases*[*to-set*]
lemmas *stream-rel-inject*[*iff*] = *stream.rel-inject*[*to-set*]

lemma *stream-rel-single-valued*[*iff*]: *single-valued* (⟨*A*⟩ *stream-rel*) ⟷ *single-valued* *A*
⟨*proof*⟩

lemmas *stream-rel-simps*[*simp*] =
stream.rel-eq[*unfolded pred-Id*, *THEN IdD*, *to-set*]
stream.rel-eq-onp[*to-set*]
stream.rel-conversep[*to-set*]
stream.rel-compp[*to-set*]

lemmas *stream-rel-param*[*param*] =
stream.ctr-transfer[*to-set*]
stream.sel-transfer[*to-set*]
stream.pred-transfer[*unfolded pred-bool-Id*, *to-set*, *folded pred-stream-streamsp*]
stream.rel-transfer[*unfolded pred-bool-Id*, *to-set*]
stream.map-transfer[*to-set*]
stream.set-transfer[*to-set*]
stream.case-transfer[*to-set*]
stream.corec-transfer[*unfolded pred-bool-Id*, *to-set*]

lemma *stream-Rangep-rel*: *Rangep* (*stream-all2 R*) = *pred-stream* (*Rangep R*)
⟨*proof*⟩

lemmas *stream-rel-domain*[*simp*] = *stream.Domainp-rel*[*to-set*]
lemmas *stream-rel-range*[*simp*] = *stream-Rangep-rel*[*to-set*]

lemma *stream-param*[*param*]:
assumes(*HOL.eq*, *HOL.eq*) ∈ *R* → *R* → *bool-rel*
shows (*HOL.eq*, *HOL.eq*) ∈ ⟨*R*⟩ *stream-rel* → ⟨*R*⟩ *stream-rel* → *bool-rel*
⟨*proof*⟩

lemmas *szip-param*[*param*] = *szip-transfer*[*to-set*]
lemmas *siterate-param*[*param*] = *siterate-transfer*[*to-set*]
lemmas *sscan-param*[*param*] = *sscan.transfer*[*to-set*]

lemma *streams-param*[*param*]: (*streams*, *streams*) ∈ ⟨*A*⟩ *set-rel* → ⟨⟨*A*⟩ *stream-rel*⟩ *set-rel*
⟨*proof*⟩

lemma *holds-param*[*param*]: (*holds*, *holds*) ∈ (*A* → *bool-rel*) → (⟨*A*⟩ *stream-rel* → *bool-rel*)
⟨*proof*⟩
lemma *HLD-param*[*param*]:
assumes *single-valued A single-valued* (A^{-1})

shows $(HLD, HLD) \in \langle A\rangle$ *set-rel* $\rightarrow \langle A\rangle$ *stream-rel* $\rightarrow$ *bool-rel*
$\langle proof\rangle$
lemma *ev-param*[*param*]: $(ev, ev) \in (\langle A\rangle$ *stream-rel* $\rightarrow$ *bool-rel*$) \rightarrow (\langle A\rangle$ *stream-rel* $\rightarrow$ *bool-rel*$)$
$\langle proof\rangle$
lemma *alw-param*[*param*]: $(alw, alw) \in (\langle A\rangle$ *stream-rel* $\rightarrow$ *bool-rel*$) \rightarrow (\langle A\rangle$ *stream-rel* $\rightarrow$ *bool-rel*$)$
$\langle proof\rangle$

25.7 Functional Relations

lemma *br-set-rel*: $\langle br\ f\ P\rangle$ *set-rel* $=$ *br* (*image f*) (λ *A*. *Ball A P*)
$\langle proof\rangle$

lemma *br-list-rel*: $\langle br\ f\ P\rangle$ *list-rel* $=$ *br* (*map f*) (*list-all P*)
$\langle proof\rangle$

lemma *br-list-set-rel*: $\langle br\ f\ P\rangle$ *list-set-rel* $=$ *br* (*set* $\circ$ *map f*) (λ *s*. *list-all P s* $\wedge$ *distinct* (*map f s*))
$\langle proof\rangle$

lemma *br-fun-rel1*: *Id* $\rightarrow$ *br f P* $=$ *br* (*comp f*) (*All* $\circ$ *comp P*)
$\langle proof\rangle$

term *set* $\circ$ *map f* $\circ$ *map g* $\circ$ *map h*

term *set* $\circ$ *sort*

end
theory *Acceptance-Refine*
imports *Acceptance Refine*
begin

abbreviation (*input*) *pred-rel A* $\equiv$ *A* $\rightarrow$ *bool-rel*
abbreviation (*input*) *rabin-rel A* $\equiv$ *pred-rel A* $\times_r$ *pred-rel A*

lemma *rabin-param*[*param*]: $(rabin, rabin) \in$ *rabin-rel A* $\rightarrow$ *pred-rel* $(\langle A\rangle$ *stream-rel*$)$
$\langle proof\rangle$
lemma *gen-param*[*param*]: $(gen, gen) \in (A \rightarrow$ *pred-rel B*$) \rightarrow (\langle A\rangle$ *list-rel* $\rightarrow$ *pred-rel B*$)$

⟨*proof*⟩
lemma *cogen-param*[*param*]: (*cogen*, *cogen*) ∈ (*A* → *pred-rel B*) → (⟨*A*⟩ *list-rel* → *pred-rel B*)
⟨*proof*⟩

end

26 Refinement for Transition Systems

theory *Transition-System-Refine*
imports
Transition-System
Transition-System-Extra
../Basic/Refine
begin

lemma *path-param*[*param*]: (*transition-system.path*, *transition-system.path*) ∈ (*T* → *S* → *S*) → (*T* → *S* → *bool-rel*) → ⟨*T*⟩ *list-rel* → *S* → *bool-rel*
⟨*proof*⟩
lemma *run-param*[*param*]: (*transition-system.run*, *transition-system.run*) ∈ (*T* → *S* → *S*) → (*T* → *S* → *bool-rel*) → ⟨*T*⟩ *stream-rel* → *S* → *bool-rel*
⟨*proof*⟩

lemma *paths-param*[*param*]:
assumes [*param*]: (*exa*, *exb*) ∈ *T* → *S* → *S*
assumes (*transition-system.enableds ena*, *transition-system.enableds enb*) ∈ *S* → ⟨*T*⟩ *set-rel*
shows (*transition-system.paths exa ena*, *transition-system.paths exb enb*) ∈ *S* → ⟨⟨*T*⟩ *list-rel*⟩ *set-rel*
⟨*proof*⟩
lemma *runs-param*[*param*]:
assumes (*exa*, *exb*) ∈ *T* → *S* → *S*
assumes (*transition-system.enableds ena*, *transition-system.enableds enb*) ∈ *S* → ⟨*T*⟩ *set-rel*
shows (*transition-system.runs exa ena*, *transition-system.runs exb enb*) ∈ *S* → ⟨⟨*T*⟩ *stream-rel*⟩ *set-rel*
⟨*proof*⟩

end

27 Relations on Deterministic Rabin Automata

theory *DRA-Refine*
imports
DRA
../../Basic/Acceptance-Refine
../../Transition-Systems/Transition-System-Refine
begin

definition *dra-rel* :: $('label_1 \times 'label_2)$ *set* $\Rightarrow$ $('state_1 \times 'state_2)$ *set* $\Rightarrow$
$(('label_1,\ 'state_1)$ *dra* $\times$ $('label_2,\ 'state_2)$ *dra*$)$ *set* **where**
[*to-relAPP*]: *dra-rel* $L\ S \equiv \{(A_1,\ A_2).$
$($*alphabet* A_1, *alphabet* $A_2) \in \langle L\rangle$ *set-rel* $\wedge$
$($*initial* A_1, *initial* $A_2) \in S\ \wedge$
$($*transition* A_1, *transition* $A_2) \in L \to S \to S\ \wedge$
$($*condition* A_1, *condition* $A_2) \in \langle$*rabin-rel* $S\rangle$ *list-rel*$\}$

lemma *dra-param*[*param*]:
$($*dra*, *dra*$) \in \langle L\rangle$ *set-rel* $\to S \to (L \to S \to S) \to \langle$*rabin-rel* $S\rangle$ *list-rel* $\to$
$\langle L,\ S\rangle$ *dra-rel*
$($*alphabet*, *alphabet*$) \in \langle L,\ S\rangle$ *dra-rel* $\to \langle L\rangle$ *set-rel*
$($*initial*, *initial*$) \in \langle L,\ S\rangle$ *dra-rel* $\to S$
$($*transition*, *transition*$) \in \langle L,\ S\rangle$ *dra-rel* $\to L \to S \to S$
$($*condition*, *condition*$) \in \langle L,\ S\rangle$ *dra-rel* $\to \langle$*rabin-rel* $S\rangle$ *list-rel*
$\langle$*proof*$\rangle$

lemma *dra-rel-id*[*simp*]: $\langle Id,\ Id\rangle$ *dra-rel* $= Id$ $\langle$*proof*$\rangle$
lemma *dra-rel-comp*[*trans*]:
assumes [*param*]: $(A,\ B) \in \langle L_1,\ S_1\rangle$ *dra-rel* $(B,\ C) \in \langle L_2,\ S_2\rangle$ *dra-rel*
shows $(A,\ C) \in \langle L_1\ O\ L_2,\ S_1\ O\ S_2\rangle$ *dra-rel*
$\langle$*proof*$\rangle$
lemma *dra-rel-converse*[*simp*]: $(\langle L,\ S\rangle$ *dra-rel*$)^{-1} = \langle L^{-1},\ S^{-1}\rangle$ *dra-rel*
$\langle$*proof*$\rangle$

lemma *dra-rel-eq*: $(A,\ A) \in \langle$*Id-on* $($*alphabet* $A)$, *Id-on* $($*nodes* $A)\rangle$ *dra-rel*
$\langle$*proof*$\rangle$

lemma *enableds-param*[*param*]: $($*dra.enableds*, *dra.enableds*$) \in \langle L,\ S\rangle$ *dra-rel* $\to$
$S \to \langle L\rangle$ *set-rel*
$\langle$*proof*$\rangle$
lemma *paths-param*[*param*]: $($*dra.paths*, *dra.paths*$) \in \langle L,\ S\rangle$ *dra-rel* $\to S \to \langle\langle L\rangle$
list-rel$\rangle$ *set-rel*
$\langle$*proof*$\rangle$
lemma *runs-param*[*param*]: $($*dra.runs*, *dra.runs*$) \in \langle L,\ S\rangle$ *dra-rel* $\to S \to \langle\langle L\rangle$
stream-rel$\rangle$ *set-rel*
$\langle$*proof*$\rangle$

lemma *reachable-param*[*param*]: $($*reachable*, *reachable*$) \in \langle L,\ S\rangle$ *dra-rel* $\to S \to$
$\langle S\rangle$ *set-rel*
$\langle$*proof*$\rangle$
lemma *nodes-param*[*param*]: $($*nodes*, *nodes*$) \in \langle L,\ S\rangle$ *dra-rel* $\to \langle S\rangle$ *set-rel*
$\langle$*proof*$\rangle$

lemma *language-param*[*param*]: $($*language*, *language*$) \in \langle L,\ S\rangle$ *dra-rel* $\to \langle\langle L\rangle$
stream-rel$\rangle$ *set-rel*
$\langle$*proof*$\rangle$

end

28 Implementation

theory *Implement*
imports
  HOL−Library.Monad-Syntax
  Collections.Refine-Dflt
  Refine
begin

28.1 Syntax

  no-syntax *-do-let* :: [*pttrn*, ′*a*] ⇒ *do-bind* ((*2let* - =/ -) [*1000*, *13*] *13*)
  syntax *-do-let* :: [*pttrn*, ′*a*] ⇒ *do-bind* ((*2let* - =/ -) *13*)

28.2 Monadic Refinement

  lemmas [*refine*] = *plain-nres-relI*

  lemma *vcg0*:
    assumes $(f, g) \in \langle Id\rangle$ *nres-rel*
    shows $g \leq h \Longrightarrow f \leq h$
    ⟨*proof*⟩
  lemma *vcg1*:
    assumes $(f, g) \in Id \rightarrow \langle Id\rangle$ *nres-rel*
    shows $g\ x \leq h\ x \Longrightarrow f\ x \leq h\ x$
    ⟨*proof*⟩
  lemma *vcg2*:
    assumes $(f, g) \in Id \rightarrow Id \rightarrow \langle Id\rangle$ *nres-rel*
    shows $g\ x\ y \leq h\ x\ y \Longrightarrow f\ x\ y \leq h\ x\ y$
    ⟨*proof*⟩

  lemma *RETURN-nres-relD*:
    assumes (*RETURN* x, *RETURN* y) $\in \langle A\rangle$ *nres-rel*
    shows $(x, y) \in A$
    ⟨*proof*⟩

  lemma *FOREACH-rule-insert*:
    assumes *finite* S
    assumes I {} s
    assumes $\bigwedge s.\ I\ S\ s \Longrightarrow P\ s$
    assumes $\bigwedge T\ x\ s.\ T \subseteq S \Longrightarrow I\ T\ s \Longrightarrow x \in S \Longrightarrow x \notin T \Longrightarrow f\ x\ s \leq$ *SPEC*
(I (*insert* x T))
    shows *FOREACH* S f s $\leq$ *SPEC* P
  ⟨*proof*⟩
  lemma *FOREACH-rule-map*:
    assumes *finite* (*dom* g)
    assumes I *Map.empty* s

assumes $\bigwedge$ *s*. *I g s* $\Longrightarrow$ *P s*
assumes $\bigwedge$ *h k v s*. *h* $\subseteq_m$ *g* $\Longrightarrow$ *I h s* $\Longrightarrow$ *g k* = *Some v* $\Longrightarrow$ *k* $\notin$ *dom h* $\Longrightarrow$
f (*k*, *v*) *s* $\leq$ *SPEC* (*I* (*h* (*k* $\mapsto$ *v*)))
shows *FOREACH* (*map-to-set g*) *f s* $\leq$ *SPEC P*
⟨*proof*⟩
lemma *FOREACH-rule-insert-eq*:
assumes *finite S*
assumes *X* {} = *s*
assumes *X S* = *t*
assumes $\bigwedge$ *T x*. *T* $\subseteq$ *S* $\Longrightarrow$ *x* $\in$ *S* $\Longrightarrow$ *x* $\notin$ *T* $\Longrightarrow$ *f x* (*X T*) $\leq$ *SPEC* (*HOL.eq* (*X* (*insert x T*)))
shows *FOREACH S f s* $\leq$ *SPEC* (*HOL.eq t*)
⟨*proof*⟩
lemma *FOREACH-rule-map-eq*:
assumes *finite* (*dom g*)
assumes *X Map.empty* = *s*
assumes *X g* = *t*
assumes $\bigwedge$ *h k v*. *h* $\subseteq_m$ *g* $\Longrightarrow$ *g k* = *Some v* $\Longrightarrow$ *k* $\notin$ *dom h* $\Longrightarrow$
f (*k*, *v*) (*X h*) $\leq$ *SPEC* (*HOL.eq* (*X* (*h* (*k* $\mapsto$ *v*))))
shows *FOREACH* (*map-to-set g*) *f s* $\leq$ *SPEC* (*HOL.eq t*)
⟨*proof*⟩

lemma *FOREACH-rule-map-map*: (*FOREACH* (*map-to-set m*) (λ (*k*, *v*). *F k* (*f k v*)),
FOREACH (*map-to-set* (λ *k*. *map-option* (*f k*) (*m k*))) (λ (*k*, *v*). *F k v*)) $\in$ *Id* $\rightarrow$ ⟨*Id*⟩ *nres-rel*
⟨*proof*⟩

28.3 Implementations for Sets Represented by Lists

lemma *list-set-rel-Id-on*[*simp*]: ⟨*Id-on A*⟩ *list-set-rel* = ⟨*Id*⟩ *list-set-rel* $\cap$ *UNIV* $\times$ *Pow A*
⟨*proof*⟩

lemma *list-set-card*[*param*]: (*length*, *card*) $\in$ ⟨*A*⟩ *list-set-rel* $\rightarrow$ *nat-rel*
⟨*proof*⟩
lemma *list-set-insert*[*param*]:
assumes *y* $\notin$ *Y*
assumes (*x*, *y*) $\in$ *A* (*xs*, *Y*) $\in$ ⟨*A*⟩ *list-set-rel*
shows (*x* # *xs*, *insert y Y*) $\in$ ⟨*A*⟩ *list-set-rel*
⟨*proof*⟩
lemma *list-set-union*[*param*]:
assumes *X* $\cap$ *Y* = {}
assumes (*xs*, *X*) $\in$ ⟨*A*⟩ *list-set-rel* (*ys*, *Y*) $\in$ ⟨*A*⟩ *list-set-rel*
shows (*xs* @ *ys*, *X* $\cup$ *Y*) $\in$ ⟨*A*⟩ *list-set-rel*
⟨*proof*⟩
lemma *list-set-Union*[*param*]:
assumes $\bigwedge$ *X Y*. *X* $\in$ *S* $\Longrightarrow$ *Y* $\in$ *S* $\Longrightarrow$ *X* $\neq$ *Y* $\Longrightarrow$ *X* $\cap$ *Y* = {}
assumes (*xs*, *S*) $\in$ ⟨⟨*A*⟩ *list-set-rel*⟩ *list-set-rel*

shows $(concat\ xs,\ Union\ S) \in \langle A\rangle\ list\text{-}set\text{-}rel$
$\langle proof\rangle$
lemma *list-set-image*[*param*]:
assumes *inj-on g S*
assumes $(f,\ g) \in A \to B\ (xs,\ S) \in \langle A\rangle\ list\text{-}set\text{-}rel$
shows $(map\ f\ xs,\ g\ `\ S) \in \langle B\rangle\ list\text{-}set\text{-}rel$
$\langle proof\rangle$
lemma *list-set-bind*[*param*]:
assumes $\bigwedge x\ y.\ x \in S \Longrightarrow y \in S \Longrightarrow x \neq y \Longrightarrow g\ x \cap g\ y = \{\}$
assumes $(xs,\ S) \in \langle A\rangle\ list\text{-}set\text{-}rel\ (f,\ g) \in A \to \langle B\rangle\ list\text{-}set\text{-}rel$
shows $(xs \ggg f,\ S \ggg g) \in \langle B\rangle\ list\text{-}set\text{-}rel$
$\langle proof\rangle$

28.4 Autoref Setup

lemma *dflt-ahm-rel-finite-nat*: *finite-map-rel* $(\langle nat\text{-}rel,\ V\rangle\ dflt\text{-}ahm\text{-}rel)$ $\langle proof\rangle$

context
begin

interpretation *autoref-syn* $\langle proof\rangle$
lemma [*autoref-op-pat*]: $(Some \circ f)\ |`\ X \equiv OP\ (\lambda\ f\ X.\ (Some \circ f)\ |`\ X)\ f\ X$
$\langle proof\rangle$
lemma [*autoref-op-pat*]: $\bigcup(m\ `\ S) \equiv OP\ (\lambda S\ m.\ \bigcup(m\ `\ S))\ S\ m$ $\langle proof\rangle$

definition *gen-UNION* **where**
$gen\text{-}UNION\ tol\ emp\ un\ X\ f \equiv fold\ (un \circ f)\ (tol\ X)\ emp$

lemma *gen-UNION*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *to-list*: *SIDE-GEN-ALGO* (*is-set-to-list A Rs1 tol*)
assumes *empty*: $GEN\text{-}OP\ emp\ \{\}\ (\langle B\rangle\ Rs3)$
assumes *union*: $GEN\text{-}OP\ un\ union\ (\langle B\rangle\ Rs2 \to \langle B\rangle\ Rs3 \to \langle B\rangle\ Rs3)$
shows $(gen\text{-}UNION\ tol\ emp\ un,\ \lambda A\ f.\ \bigcup\ (f\ `\ A)) \in \langle A\rangle\ Rs1 \to (A \to \langle B\rangle\ Rs2) \to \langle B\rangle\ Rs3$
$\langle proof\rangle$

definition *gen-Image* **where**
$gen\text{-}Image\ tol1\ mem2\ emp3\ ins3\ X\ Y \equiv fold$
$(\lambda\ (a,\ b).\ if\ mem2\ a\ Y\ then\ ins3\ b\ else\ id)\ (tol1\ X)\ emp3$

lemma *gen-Image*[*autoref-rules*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *to-list*: $SIDE\text{-}GEN\text{-}ALGO\ (is\text{-}set\text{-}to\text{-}list\ (A \times_r B)\ Rs1\ tol1)$
assumes *member*: $GEN\text{-}OP\ mem2\ (\in)\ (A \to \langle A\rangle\ Rs2 \to bool\text{-}rel)$
assumes *empty*: $GEN\text{-}OP\ emp3\ \{\}\ (\langle B\rangle\ Rs3)$
assumes *insert*: $GEN\text{-}OP\ ins3\ Set.insert\ (B \to \langle B\rangle\ Rs3 \to \langle B\rangle\ Rs3)$
shows $(gen\text{-}Image\ tol1\ mem2\ emp3\ ins3,\ Image) \in \langle A \times_r B\rangle\ Rs1 \to \langle A\rangle\ Rs2 \to \langle B\rangle\ Rs3$

$\langle proof \rangle$

lemma *list-set-union-autoref*[*autoref-rules*]:
  assumes *PRIO-TAG-OPTIMIZATION*
  assumes *SIDE-PRECOND-OPT* $(a' \cap b' = \{\})$
  assumes $(a,\ a') \in \langle R\rangle$ *list-set-rel*
  assumes $(b,\ b') \in \langle R\rangle$ *list-set-rel*
  shows $(a\ @\ b,$
    $(OP\ union ::: \langle R\rangle\ \textit{list-set-rel} \to \langle R\rangle\ \textit{list-set-rel} \to \langle R\rangle\ \textit{list-set-rel})\ \$\ a'\ \$\ b')$
$\in$
    $\langle R\rangle$ *list-set-rel*
  $\langle proof \rangle$
lemma *list-set-image-autoref*[*autoref-rules*]:
  assumes *PRIO-TAG-OPTIMIZATION*
  assumes *INJ*: *SIDE-PRECOND-OPT* (*inj-on f s*)
  assumes $\bigwedge xi\ x.\ (xi,\ x) \in Ra \Longrightarrow x \in s \Longrightarrow (fi\ xi,\ f\ \$\ x) \in Rb$
  assumes *LP*: $(l,s) \in \langle Ra\rangle$*list-set-rel*
  shows (*map fi l*,
    $(OP\ image ::: (Ra \to Rb) \to \langle Ra\rangle\ \textit{list-set-rel} \to \langle Rb\rangle\ \textit{list-set-rel})\ \$\ f\ \$\ s) \in$
    $\langle Rb\rangle$ *list-set-rel*
$\langle proof \rangle$
lemma *list-set-UNION-autoref*[*autoref-rules*]:
  assumes *PRIO-TAG-OPTIMIZATION*
  assumes *SIDE-PRECOND-OPT* $(\forall\ x \in S.\ \forall\ y \in S.\ x \neq y \longrightarrow g\ x \cap g\ y =$
$\{\})$
  assumes $(xs,\ S) \in \langle A\rangle$ *list-set-rel* $(f,\ g) \in A \to \langle B\rangle$ *list-set-rel*
  shows $(xs \ggg= f,$
    $(OP\ (\lambda A\ f.\ \bigcup\ (f\ `\ A)) ::: \langle A\rangle\ \textit{list-set-rel} \to (A \to \langle B\rangle\ \textit{list-set-rel}) \to \langle B\rangle$
list-set-rel$)\ \$\ S\ \$\ g) \in$
    $\langle B\rangle$ *list-set-rel*
  $\langle proof \rangle$

definition *gen-equals* **where**
  gen-equals ball lu eq f g $\equiv$
    ball f $(\lambda\ (k,\ v).$ *rel-option eq* $(lu\ k\ g)\ (Some\ v)) \wedge$
    ball g $(\lambda\ (k,\ v).$ *rel-option eq* $(lu\ k\ f)\ (Some\ v))$

lemma *gen-equals*[*autoref-rules*]:
  assumes *PRIO-TAG-GEN-ALGO*
  assumes *BALL*: *GEN-OP ball op-map-ball* $(\langle Rk,\ Rv\rangle\ Rm \to (Rk \times_r Rv \to$
bool-rel$) \to$ *bool-rel*$)$
  assumes *LU*: *GEN-OP lu op-map-lookup* $(Rk \to \langle Rk,\ Rv\rangle\ Rm \to \langle Rv\rangle$
option-rel$)$
  assumes *EQ*: *GEN-OP eq HOL.eq* $(Rv \to Rv \to$ *bool-rel*$)$
  shows (*gen-equals ball lu eq*, *HOL.eq*$) \in \langle Rk,\ Rv\rangle\ Rm \to \langle Rk,\ Rv\rangle\ Rm \to$
bool-rel
$\langle proof \rangle$

definition *op-set-enumerate* :: $'a$ *set* $\Rightarrow$ ($'a \rightharpoonup$ *nat*) *nres* **where**
op-set-enumerate S $\equiv$ *SPEC* (λ *f*. *dom f* = *S* $\wedge$ *inj-on f S*)

lemma [*autoref-itype*]: *op-set-enumerate* $::_i$ $\langle A\rangle_i$ *i-set* $\rightarrow_i$ $\langle\langle A,$ *i-nat*$\rangle_i$ *i-map*$\rangle_i$ *i-nres* $\langle proof\rangle$
lemma [*autoref-hom*]: *CONSTRAINT op-set-enumerate* ($\langle A\rangle$ *Rs* $\rightarrow$ $\langle\langle A$, *nat-rel*$\rangle$ *Rm*$\rangle$ *nres-rel*) $\langle proof\rangle$

definition *gen-enumerate* **where**
gen-enumerate tol upd emp S $\equiv$ *snd* (*fold* (λ *x* (*k*, *m*). (*Suc k*, *upd x k m*)) (*tol S*) (*0*, *emp*))

lemma *gen-enumerate*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *to-list*: *SIDE-GEN-ALGO* (*is-set-to-list A Rs tol*)
assumes *empty*: *GEN-OP emp op-map-empty* ($\langle A$, *nat-rel*$\rangle$ *Rm*)
assumes *update*: *GEN-OP upd op-map-update* ($A \rightarrow$ *nat-rel* $\rightarrow$ $\langle A$, *nat-rel*$\rangle$ *Rm* $\rightarrow$ $\langle A$, *nat-rel*$\rangle$ *Rm*)
shows (λ *S*. *RETURN* (*gen-enumerate tol upd emp S*), *op-set-enumerate*) $\in$
$\langle A\rangle$ *Rs* $\rightarrow$ $\langle\langle A$, *nat-rel*$\rangle$ *Rm*$\rangle$ *nres-rel*
$\langle proof\rangle$

lemma *gen-enumerate-it-to-list*[*refine-transfer-post-simp*]:
gen-enumerate (*it-to-list it*) =
(λ *upd emp S*. *snd* (*foldli* (*it-to-list it S*) (λ -. *True*)
(λ *x s*. *case s of* (*k*, *m*) $\Rightarrow$ (*Suc k*, *upd x k m*)) (*0*, *emp*)))
$\langle proof\rangle$

definition *gen-build* **where**
gen-build tol upd emp f X $\equiv$ *fold* (λ *x*. *upd x* (*f x*)) (*tol X*) *emp*

lemma *gen-build*[*autoref-rules*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *to-list*: *SIDE-GEN-ALGO* (*is-set-to-list A Rs tol*)
assumes *empty*: *GEN-OP emp op-map-empty* ($\langle A, B\rangle$ *Rm*)
assumes *update*: *GEN-OP upd op-map-update* ($A \rightarrow B \rightarrow \langle A, B\rangle$ *Rm* $\rightarrow \langle A, B\rangle$ *Rm*)
shows (λ *f X*. *gen-build tol upd emp f X*, λ *f X*. (*Some* $\circ$ *f*) |‘ *X*) $\in$
($A \rightarrow B$) $\rightarrow$ $\langle A\rangle$ *Rs* $\rightarrow$ $\langle A, B\rangle$ *Rm*
$\langle proof\rangle$

definition *to-list it s* $\equiv$ *it s top Cons Nil*

lemma *map2set-to-list*:
assumes *GEN-ALGO-tag* (*is-map-to-list Rk unit-rel R it*)
shows *is-set-to-list Rk* (*map2set-rel R*) (*to-list* (*map-iterator-dom* $\circ$ (*foldli* $\circ$ *it*)))
$\langle proof\rangle$

lemma *CAST-to-list*[*autoref-rules-raw*]:
  assumes *PRIO-TAG-GEN-ALGO*
  assumes *SIDE-GEN-ALGO* (*is-set-to-list A Rs tol*)
  shows (*tol*, *CAST*) $\in$ $\langle A\rangle$ *Rs* $\rightarrow$ $\langle A\rangle$ *list-set-rel*
  $\langle$*proof*$\rangle$

lemma *param-foldli*:
  assumes (*xs*, *ys*) $\in$ $\langle Ra\rangle$ *list-rel*
  assumes (*c*, *d*) $\in$ *Rs* $\rightarrow$ *bool-rel*
  assumes $\bigwedge$ *x y*. (*x*, *y*) $\in$ *Ra* $\Longrightarrow$ *x* $\in$ *set xs* $\Longrightarrow$ *y* $\in$ *set ys* $\Longrightarrow$ (*f x*, *g y*) $\in$ *Rs* $\rightarrow$ *Rs*
  assumes (*a*, *b*) $\in$ *Rs*
  shows (*foldli xs c f a*, *foldli ys d g b*) $\in$ *Rs*
$\langle$*proof*$\rangle$
lemma *det-fold-sorted-set*:
  assumes *1*: *det-fold-set ordR c' f' σ' result*
  assumes *2*: *is-set-to-sorted-list ordR Rk Rs tsl*
  assumes *SREF*[*param*]: (*s*,*s'*)$\in\langle Rk\rangle$*Rs*
  assumes [*param*]: (*c*,*c'*)$\in$*Rσ*$\rightarrow$*Id*
  assumes [*param*]: $\bigwedge$ *x y*. (*x*, *y*) $\in$ *Rk* $\Longrightarrow$ *y* $\in$ *s'* $\Longrightarrow$ (*f x*,*f' y*)$\in$*Rσ* $\rightarrow$ *Rσ*
  assumes [*param*]: (*σ*,*σ'*)$\in$*Rσ*
  shows (*foldli* (*tsl s*) *c f σ*, *result s'*) $\in$ *Rσ*
$\langle$*proof*$\rangle$
lemma *det-fold-set*:
  assumes *det-fold-set* (λ- -. *True*) *c' f' σ' result*
  assumes *is-set-to-list Rk Rs tsl*
  assumes (*s*,*s'*)$\in\langle Rk\rangle$*Rs*
  assumes (*c*,*c'*)$\in$*Rσ*$\rightarrow$*Id*
  assumes $\bigwedge$ *x y*. (*x*, *y*) $\in$ *Rk* $\Longrightarrow$ *y* $\in$ *s'* $\Longrightarrow$ (*f x*, *f' y*)$\in$*Rσ* $\rightarrow$ *Rσ*
  assumes (*σ*,*σ'*)$\in$*Rσ*
  shows (*foldli* (*tsl s*) *c f σ*, *result s'*) $\in$ *Rσ*
  $\langle$*proof*$\rangle$
lemma *gen-image*[*autoref-rules-raw*]:
  assumes *PRIO-TAG-GEN-ALGO*
  assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs1 it1*)
  assumes *INS*: *GEN-OP ins2 Set.insert* (*Rk'*$\rightarrow\langle$*Rk'*$\rangle$*Rs2*$\rightarrow\langle$*Rk'*$\rangle$*Rs2*)
  assumes *EMPTY*: *GEN-OP empty2* {} ($\langle$*Rk'*$\rangle$*Rs2*)
  assumes $\bigwedge$ *xi x*. (*xi*, *x*) $\in$ *Rk* $\Longrightarrow$ *x* $\in$ *s* $\Longrightarrow$ (*fi xi*, *f* \$ *x*) $\in$ *Rk'*
  assumes (*l*, *s*) $\in$ $\langle Rk\rangle$*Rs1*
  shows (*gen-image* (λ *x*. *foldli* (*it1 x*)) *empty2 ins2 fi l*,
   (*OP image* ::: (*Rk*$\rightarrow$*Rk'*) $\rightarrow$ ($\langle Rk\rangle$*Rs1*) $\rightarrow$ ($\langle$*Rk'*$\rangle$*Rs2*)) \$ *f* \$ *s*) $\in$ ($\langle$*Rk'*$\rangle$*Rs2*)
$\langle$*proof*$\rangle$

end

end

29 Implementation of Deterministic Rabin Automata

theory *DRA-Implement*
imports
  DRA-Refine
  ../../Basic/Implement
begin

  datatype (*'label*, *'state*) *drai* = *drai*
    (*alphabeti*: *'label list*)
    (*initiali*: *'state*)
    (*transitioni*: *'label* ⇒ *'state* ⇒ *'state*)
    (*conditioni*: *'state rabin gen*)

  definition *drai-rel* :: ($'label_1$ × $'label_2$) *set* ⇒ ($'state_1$ × $'state_2$) *set* ⇒
    (($'label_1$, $'state_1$) *drai* × ($'label_2$, $'state_2$) *drai*) *set* **where**
    [*to-relAPP*]: *drai-rel L S* ≡ {(A_1, A_2).
      (*alphabeti* A_1, *alphabeti* A_2) ∈ ⟨*L*⟩ *list-rel* ∧
      (*initiali* A_1, *initiali* A_2) ∈ *S* ∧
      (*transitioni* A_1, *transitioni* A_2) ∈ *L* → *S* → *S* ∧
      (*conditioni* A_1, *conditioni* A_2) ∈ ⟨*rabin-rel S*⟩ *list-rel*}

  lemma *drai-param*[*param*]:
    (*drai*, *drai*) ∈ ⟨*L*⟩ *list-rel* → *S* → (*L* → *S* → *S*) →
      ⟨*rabin-rel S*⟩ *list-rel* → ⟨*L*, *S*⟩ *drai-rel*
    (*alphabeti*, *alphabeti*) ∈ ⟨*L*, *S*⟩ *drai-rel* → ⟨*L*⟩ *list-rel*
    (*initiali*, *initiali*) ∈ ⟨*L*, *S*⟩ *drai-rel* → *S*
    (*transitioni*, *transitioni*) ∈ ⟨*L*, *S*⟩ *drai-rel* → *L* → *S* → *S*
    (*conditioni*, *conditioni*) ∈ ⟨*L*, *S*⟩ *drai-rel* → ⟨*rabin-rel S*⟩ *list-rel*
    ⟨*proof*⟩

  definition *drai-dra-rel* :: ($'label_1$ × $'label_2$) *set* ⇒ ($'state_1$ × $'state_2$) *set* ⇒
    (($'label_1$, $'state_1$) *drai* × ($'label_2$, $'state_2$) *dra*) *set* **where**
    [*to-relAPP*]: *drai-dra-rel L S* ≡ {(A_1, A_2).
      (*alphabeti* A_1, *alphabet* A_2) ∈ ⟨*L*⟩ *list-set-rel* ∧
      (*initiali* A_1, *initial* A_2) ∈ *S* ∧
      (*transitioni* A_1, *transition* A_2) ∈ *L* → *S* → *S* ∧
      (*conditioni* A_1, *condition* A_2) ∈ ⟨*rabin-rel S*⟩ *list-rel*}

  lemma *drai-dra-param*[*param*, *autoref-rules*]:
    (*drai*, *dra*) ∈ ⟨*L*⟩ *list-set-rel* → *S* → (*L* → *S* → *S*) →
      ⟨*rabin-rel S*⟩ *list-rel* → ⟨*L*, *S*⟩ *drai-dra-rel*
    (*alphabeti*, *alphabet*) ∈ ⟨*L*, *S*⟩ *drai-dra-rel* → ⟨*L*⟩ *list-set-rel*
    (*initiali*, *initial*) ∈ ⟨*L*, *S*⟩ *drai-dra-rel* → *S*
    (*transitioni*, *transition*) ∈ ⟨*L*, *S*⟩ *drai-dra-rel* → *L* → *S* → *S*
    (*conditioni*, *condition*) ∈ ⟨*L*, *S*⟩ *drai-dra-rel* → ⟨*rabin-rel S*⟩ *list-rel*
    ⟨*proof*⟩

  definition *drai-dra* :: (*'label*, *'state*) *drai* ⇒ (*'label*, *'state*) *dra* **where**

drai-dra A ≡ *dra* (*set* (*alphabeti A*)) (*initiali A*) (*transitioni A*) (*conditioni A*)
definition *drai-invar* :: ('*label*, '*state*) *drai* ⇒ *bool* **where**
drai-invar A ≡ *distinct* (*alphabeti A*)

lemma *drai-dra-id-param*[*param*]: (*drai-dra*, *id*) ∈ ⟨*L*, *S*⟩ *drai-dra-rel* → ⟨*L*, *S*⟩ *dra-rel*
⟨*proof*⟩

lemma *drai-dra-br*: ⟨*Id*, *Id*⟩ *drai-dra-rel* = *br drai-dra drai-invar*
⟨*proof*⟩

end

30 Exploration of Deterministic Rabin Automata

theory *DRA-Nodes*
imports
DFS-Framework.Reachable-Nodes
DRA-Implement
begin

definition *dra-G* :: ('*label*, '*state*) *dra* ⇒ '*state graph-rec* **where**
dra-G A ≡ (| *g-V* = *UNIV*, *g-E* = *E-of-succ* (*successors A*), *g-V0* = {*initial A*} |)

lemma *dra-G-graph*[*simp*]: *graph* (*dra-G A*) ⟨*proof*⟩
lemma *dra-G-reachable-nodes*: *op-reachable* (*dra-G A*) = *nodes A*
⟨*proof*⟩

context
begin

interpretation *autoref-syn* ⟨*proof*⟩

lemma *dra-G-ahs*: *dra-G A* = (| *g-V* = *UNIV*, *g-E* = *E-of-succ* (λ *p*. *CAST* ((λ *a*. *transition A a p* ::: *S*) ' *alphabet A* ::: ⟨*S*⟩ *ahs-rel bhc*)), *g-V0* = {*initial A*} |)
⟨*proof*⟩

schematic-goal *drai-Gi*:
notes *map2set-to-list*[*autoref-ga-rules*]
fixes *S* :: ('*statei* × '*state*) *set*
assumes [*autoref-ga-rules*]: *is-bounded-hashcode S seq bhc*
assumes [*autoref-ga-rules*]: *is-valid-def-hm-size TYPE*('*statei*) *hms*
assumes [*autoref-rules*]: (*seq*, *HOL.eq*) ∈ *S* → *S* → *bool-rel*
assumes [*autoref-rules*]: (*Ai*, *A*) ∈ ⟨*L*, *S*⟩ *drai-dra-rel*
shows (*?f* :: *?'a*, *RETURN* (*dra-G A*)) ∈ *?A*
⟨*proof*⟩
concrete-definition *drai-Gi* **uses** *drai-Gi*

lemma *drai-Gi-refine*[*autoref-rules*]:
fixes S :: (*'statei* × *'state*) *set*
assumes *SIDE-GEN-ALGO* (*is-bounded-hashcode S seq bhc*)
assumes *SIDE-GEN-ALGO* (*is-valid-def-hm-size TYPE*(*'statei*) *hms*)
assumes *GEN-OP seq HOL.eq* ($S \to S \to$ *bool-rel*)
shows (*DRA-Nodes.drai-Gi seq bhc hms*, *dra-G*) ∈ ⟨*L*, *S*⟩ *drai-dra-rel* → ⟨*unit-rel*, *S*⟩ *g-impl-rel-ext*
⟨*proof*⟩

schematic-goal *dra-nodes*:
fixes S :: (*'statei* × *'state*) *set*
assumes [*simp*]: *finite* ((*g-E* (*dra-G A*))* ‘‘ *g-V0* (*dra-G A*))
assumes [*autoref-ga-rules*]: *is-bounded-hashcode S seq bhc*
assumes [*autoref-ga-rules*]: *is-valid-def-hm-size TYPE*(*'statei*) *hms*
assumes [*autoref-rules*]: (*seq*, *HOL.eq*) ∈ $S \to S \to$ *bool-rel*
assumes [*autoref-rules*]: (*Ai*, *A*) ∈ ⟨*L*, *S*⟩ *drai-dra-rel*
shows (*?f* :: *?'a*, *op-reachable* (*dra-G A*)) ∈ *?R* ⟨*proof*⟩
concrete-definition *dra-nodes* **uses** *dra-nodes*
lemma *dra-nodes-refine*[*autoref-rules*]:
fixes S :: (*'statei* × *'state*) *set*
assumes *SIDE-PRECOND* (*finite* (*nodes A*))
assumes *SIDE-GEN-ALGO* (*is-bounded-hashcode S seq bhc*)
assumes *SIDE-GEN-ALGO* (*is-valid-def-hm-size TYPE*(*'statei*) *hms*)
assumes *GEN-OP seq HOL.eq* ($S \to S \to$ *bool-rel*)
assumes (*Ai*, *A*) ∈ ⟨*L*, *S*⟩ *drai-dra-rel*
shows (*DRA-Nodes.dra-nodes seq bhc hms Ai*,
(*OP nodes* ::: ⟨*L*, *S*⟩ *drai-dra-rel* → ⟨*S*⟩ *ahs-rel bhc*) $ *A*) ∈ ⟨*S*⟩ *ahs-rel bhc*
⟨*proof*⟩

end

end

31 Explicit Deterministic Rabin Automata

theory *DRA-Explicit*
imports *DRA-Nodes*
begin

datatype (*'label*, *'state*) *drae* = *drae*
(*alphabete*: *'label set*)
(*initiale*: *'state*)
(*transitione*: (*'state* × *'label* × *'state*) *set*)
(*conditione*: (*'state set* × *'state set*) *list*)

definition *drae-rel* **where**
[*to-relAPP*]: *drae-rel L S* ≡ {(A_1, A_2).
(*alphabete* A_1, *alphabete* A_2) ∈ ⟨*L*⟩ *set-rel* ∧

$(\mathit{initiale}\ A_1,\ \mathit{initiale}\ A_2) \in S \wedge$
$(\mathit{transitione}\ A_1,\ \mathit{transitione}\ A_2) \in \langle S \times_r L \times_r S\rangle\ \mathit{set\text{-}rel} \wedge$
$(\mathit{conditione}\ A_1,\ \mathit{conditione}\ A_2) \in \langle\langle S\rangle\ \mathit{set\text{-}rel} \times_r \langle S\rangle\ \mathit{set\text{-}rel}\rangle\ \mathit{list\text{-}rel}\}$

lemma *drae-param*[*param*, *autoref-rules*]:
$(\mathit{drae},\ \mathit{drae}) \in \langle L\rangle\ \mathit{set\text{-}rel} \to S \to \langle S \times_r L \times_r S\rangle\ \mathit{set\text{-}rel} \to$
$\langle\langle S\rangle\ \mathit{set\text{-}rel} \times_r \langle S\rangle\ \mathit{set\text{-}rel}\rangle\ \mathit{list\text{-}rel} \to \langle L,\ S\rangle\ \mathit{drae\text{-}rel}$
$(\mathit{alphabete},\ \mathit{alphabete}) \in \langle L,\ S\rangle\ \mathit{drae\text{-}rel} \to \langle L\rangle\ \mathit{set\text{-}rel}$
$(\mathit{initiale},\ \mathit{initiale}) \in \langle L,\ S\rangle\ \mathit{drae\text{-}rel} \to S$
$(\mathit{transitione},\ \mathit{transitione}) \in \langle L,\ S\rangle\ \mathit{drae\text{-}rel} \to \langle S \times_r L \times_r S\rangle\ \mathit{set\text{-}rel}$
$(\mathit{conditione},\ \mathit{conditione}) \in \langle L,\ S\rangle\ \mathit{drae\text{-}rel} \to \langle\langle S\rangle\ \mathit{set\text{-}rel} \times_r \langle S\rangle\ \mathit{set\text{-}rel}\rangle\ \mathit{list\text{-}rel}$
⟨*proof*⟩

lemma *drae-rel-id*[*simp*]: $\langle \mathit{Id},\ \mathit{Id}\rangle\ \mathit{drae\text{-}rel} = \mathit{Id}$ ⟨*proof*⟩
lemma *drae-rel-comp*[*simp*]: $\langle L_1\ O\ L_2,\ S_1\ O\ S_2\rangle\ \mathit{drae\text{-}rel} = \langle L_1,\ S_1\rangle\ \mathit{drae\text{-}rel}\ O$
$\langle L_2,\ S_2\rangle\ \mathit{drae\text{-}rel}$
⟨*proof*⟩

consts *i-drae-scheme* :: *interface* ⇒ *interface* ⇒ *interface*

context
begin

interpretation *autoref-syn* ⟨*proof*⟩

lemma *drae-scheme-itype*[*autoref-itype*]:
$\mathit{drae} ::_i \langle L\rangle_i\ \mathit{i\text{-}set} \to_i S \to_i \langle\langle S,\ \langle L,\ S\rangle_i\ \mathit{i\text{-}prod}\rangle_i\ \mathit{i\text{-}prod}\rangle_i\ \mathit{i\text{-}set} \to_i$
$\langle\langle\langle S\rangle_i\ \mathit{i\text{-}set},\ \langle S\rangle_i\ \mathit{i\text{-}set}\rangle_i\ \mathit{i\text{-}prod}\rangle_i\ \mathit{i\text{-}list} \to_i \langle L,\ S\rangle_i\ \mathit{i\text{-}drae\text{-}scheme}$
$\mathit{alphabete} ::_i \langle L,\ S\rangle_i\ \mathit{i\text{-}drae\text{-}scheme} \to_i \langle L\rangle_i\ \mathit{i\text{-}set}$
$\mathit{initiale} ::_i \langle L,\ S\rangle_i\ \mathit{i\text{-}drae\text{-}scheme} \to_i S$
$\mathit{transitione} ::_i \langle L,\ S\rangle_i\ \mathit{i\text{-}drae\text{-}scheme} \to_i \langle\langle S,\ \langle L,\ S\rangle_i\ \mathit{i\text{-}prod}\rangle_i\ \mathit{i\text{-}prod}\rangle_i\ \mathit{i\text{-}set}$
$\mathit{conditione} ::_i \langle L,\ S\rangle_i\ \mathit{i\text{-}drae\text{-}scheme} \to_i \langle\langle\langle S\rangle_i\ \mathit{i\text{-}set},\ \langle S\rangle_i\ \mathit{i\text{-}set}\rangle_i\ \mathit{i\text{-}prod}\rangle_i\ \mathit{i\text{-}list}$
⟨*proof*⟩

end

datatype $('\mathit{label},\ '\mathit{state})\ \mathit{draei} = \mathit{draei}$
$(\mathit{alphabetei}\colon '\mathit{label}\ \mathit{list})$
$(\mathit{initialei}\colon '\mathit{state})$
$(\mathit{transitionei}\colon ('\mathit{state} \times '\mathit{label} \times '\mathit{state})\ \mathit{list})$
$(\mathit{conditionei}\colon ('\mathit{state}\ \mathit{list} \times '\mathit{state}\ \mathit{list})\ \mathit{list})$

definition *draei-rel* **where**
$[\mathit{to\text{-}relAPP}]\colon \mathit{draei\text{-}rel}\ L\ S \equiv \{(A_1,\ A_2).$
$(\mathit{alphabetei}\ A_1,\ \mathit{alphabetei}\ A_2) \in \langle L\rangle\ \mathit{list\text{-}rel} \wedge$
$(\mathit{initialei}\ A_1,\ \mathit{initialei}\ A_2) \in S \wedge$
$(\mathit{transitionei}\ A_1,\ \mathit{transitionei}\ A_2) \in \langle S \times_r L \times_r S\rangle\ \mathit{list\text{-}rel} \wedge$
$(\mathit{conditionei}\ A_1,\ \mathit{conditionei}\ A_2) \in \langle\langle S\rangle\ \mathit{list\text{-}rel} \times_r \langle S\rangle\ \mathit{list\text{-}rel}\rangle\ \mathit{list\text{-}rel}\}$

lemma *draei-param*[*param*, *autoref-rules*]:
$(draei, draei) \in \langle L\rangle$ *list-rel* $\to S \to \langle S \times_r L \times_r S\rangle$ *list-rel* $\to$
$\langle\langle S\rangle$ *list-rel* $\times_r \langle S\rangle$ *list-rel*$\rangle$ *list-rel* $\to \langle L, S\rangle$ *draei-rel*
$(alphabetei, alphabetei) \in \langle L, S\rangle$ *draei-rel* $\to \langle L\rangle$ *list-rel*
$(initialei, initialei) \in \langle L, S\rangle$ *draei-rel* $\to S$
$(transitionei, transitionei) \in \langle L, S\rangle$ *draei-rel* $\to \langle S \times_r L \times_r S\rangle$ *list-rel*
$(conditionei, conditionei) \in \langle L, S\rangle$ *draei-rel* $\to \langle\langle S\rangle$ *list-rel* $\times_r \langle S\rangle$ *list-rel*$\rangle$ *list-rel*
$\langle proof\rangle$

definition *draei-drae-rel* **where**
[*to-relAPP*]: *draei-drae-rel* $L\ S \equiv \{(A_1, A_2)$.
$(alphabetei\ A_1, alphabete\ A_2) \in \langle L\rangle$ *list-set-rel* $\wedge$
$(initialei\ A_1, initiale\ A_2) \in S \wedge$
$(transitionei\ A_1, transitione\ A_2) \in \langle S \times_r L \times_r S\rangle$ *list-set-rel* $\wedge$
$(conditionei\ A_1, conditione\ A_2) \in \langle\langle S\rangle$ *list-set-rel* $\times_r \langle S\rangle$ *list-set-rel*$\rangle$ *list-rel*$\}$

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of draei-drae-rel i-drae-scheme*]

lemma *draei-drae-param*[*param*, *autoref-rules*]:
$(draei, drae) \in \langle L\rangle$ *list-set-rel* $\to S \to \langle S \times_r L \times_r S\rangle$ *list-set-rel* $\to$
$\langle\langle S\rangle$ *list-set-rel* $\times_r \langle S\rangle$ *list-set-rel*$\rangle$ *list-rel* $\to \langle L, S\rangle$ *draei-drae-rel*
$(alphabetei, alphabete) \in \langle L, S\rangle$ *draei-drae-rel* $\to \langle L\rangle$ *list-set-rel*
$(initialei, initiale) \in \langle L, S\rangle$ *draei-drae-rel* $\to S$
$(transitionei, transitione) \in \langle L, S\rangle$ *draei-drae-rel* $\to \langle S \times_r L \times_r S\rangle$ *list-set-rel*
$(conditionei, conditione) \in \langle L, S\rangle$ *draei-drae-rel* $\to \langle\langle S\rangle$ *list-set-rel* $\times_r \langle S\rangle$ *list-set-rel*$\rangle$ *list-rel*
$\langle proof\rangle$

definition *draei-drae* **where**
draei-drae $A \equiv$ *drae* (*set* (*alphabetei* A)) (*initialei* A)
(*set* (*transitionei* A)) (*map* (*map-prod set set*) (*conditionei* A))

lemma *draei-drae-id-param*[*param*]: $(draei\text{-}drae, id) \in \langle L, S\rangle$ *draei-drae-rel* $\to \langle L, S\rangle$ *drae-rel*
$\langle proof\rangle$

abbreviation *transitions* $L\ S\ s \equiv \bigcup\ a \in L.\ \bigcup\ p \in S.\ \{p\} \times \{a\} \times \{s\ a\ p\}$
abbreviation *succs* $T\ a\ p \equiv$ *the-elem* $((T\ ``\ \{p\})\ ``\ \{a\})$

definition *wft* :: *'label set* $\Rightarrow$ *'state set* $\Rightarrow$ (*'state* $\times$ *'label* $\times$ *'state*) *set* $\Rightarrow$ *bool* **where**
wft $L\ S\ T \equiv \forall\ a \in L.\ \forall\ p \in S.$ *is-singleton* $((T\ ``\ \{p\})\ ``\ \{a\})$

lemma *wft-param*[*param*]:
assumes *bijective S bijective L*
shows $(wft, wft) \in \langle L\rangle$ *set-rel* $\to \langle S\rangle$ *set-rel* $\to \langle S \times_r L \times_r S\rangle$ *set-rel* $\to$ *bool-rel*
$\langle proof\rangle$

lemma *wft-transitions*: *wft L S* (*transitions L S s*) ⟨*proof*⟩

definition *dra-drae* **where** *dra-drae A* ≡ *drae* (*alphabet A*) (*initial A*)
(*transitions* (*alphabet A*) (*nodes A*) (*transition A*))
(*map* (λ (*P*, *Q*). (*Set.filter P* (*nodes A*), *Set.filter Q* (*nodes A*))) (*condition A*))
definition *drae-dra* **where** *drae-dra A* ≡ *dra* (*alphabete A*) (*initiale A*)
(*succs* (*transitione A*)) (*map* (λ (*I*, *F*). (λ *p*. *p* ∈ *I*, λ *p*. *p* ∈ *F*)) (*conditione A*))

lemma *set-rel-Domain-Range*[*intro*!, *simp*]: (*Domain A*, *Range A*) ∈ ⟨*A*⟩ *set-rel* ⟨*proof*⟩

lemma *dra-drae-param*[*param*]: (*dra-drae*, *dra-drae*) ∈ ⟨*L*, *S*⟩ *dra-rel* → ⟨*L*, *S*⟩ *drae-rel*
⟨*proof*⟩
lemma *drae-dra-param*[*param*]:
assumes *bijective L bijective S*
assumes *wft* (*Range L*) (*Range S*) (*transitione B*)
assumes [*param*]: (*A*, *B*) ∈ ⟨*L*, *S*⟩ *drae-rel*
shows (*drae-dra A*, *drae-dra B*) ∈ ⟨*L*, *S*⟩ *dra-rel*
⟨*proof*⟩

lemma *succs-transitions-param*[*param*]:
(*succs* ∘ *transitions L S*, *id*) ∈ (*Id-on L* → *Id-on S* → *Id-on S*) → (*Id-on L* → *Id-on S* → *Id-on S*)
⟨*proof*⟩
lemma *drae-dra-dra-drae-param*[*param*]:
((*drae-dra* ∘ *dra-drae*) *A*, *id A*) ∈ ⟨*Id-on* (*alphabet A*), *Id-on* (*nodes A*)⟩ *dra-rel*
⟨*proof*⟩

definition *draei-dra-rel* **where**
[*to-relAPP*]: *draei-dra-rel L S* ≡ {(*Ae*, *A*). (*drae-dra* (*draei-drae Ae*), *A*) ∈ ⟨*L*, *S*⟩ *dra-rel*}
lemma *draei-dra-id*[*param*]: (*drae-dra* ∘ *draei-drae*, *id*) ∈ ⟨*L*, *S*⟩ *draei-dra-rel* → ⟨*L*, *S*⟩ *dra-rel*
⟨*proof*⟩

end

32 Explore and Enumerate Nodes of Deterministic Rabin Automata

theory *DRA-Translate*
imports *DRA-Explicit*
begin

32.1 Syntax

no-syntax *-do-let* :: [*pttrn*, *'a*] ⇒ *do-bind* ((*2let* - =/ -) [*1000*, *13*] *13*)
syntax *-do-let* :: [*pttrn*, *'a*] ⇒ *do-bind* ((*2let* - =/ -) *13*)

33 Image on Explicit Automata

definition *drae-image* **where** *drae-image f A* ≡ *drae* (*alphabete A*) (*f* (*initiale A*))
((λ (*p*, *a*, *q*). (*f p*, *a*, *f q*)) ‘ *transitione A*) (*map* (*map-prod* (*image f*) (*image f*)) (*conditione A*))

lemma *drae-image-param*[*param*]: (*drae-image*, *drae-image*) ∈ (*S* → *T*) → ⟨*L*, *S*⟩ *drae-rel* → ⟨*L*, *T*⟩ *drae-rel*
⟨*proof*⟩

lemma *drae-image-id*[*simp*]: *drae-image id* = *id* ⟨*proof*⟩
lemma *drae-image-dra-drae*: *drae-image f* (*dra-drae A*) = *drae*
(*alphabet A*) (*f* (*initial A*))
(⋃ *p* ∈ *nodes A*. ⋃ *a* ∈ *alphabet A*. *f* ‘ {*p*} × {*a*} × *f* ‘ {*transition A a p*})
(*map* (λ (*P*, *Q*). (*f* ‘ {*p* ∈ *nodes A*. *P p*}, *f* ‘ {*p* ∈ *nodes A*. *Q p*})) (*condition A*))
⟨*proof*⟩

34 Exploration and Translation

definition *trans-spec* **where**
trans-spec A f ≡ ⋃ *p* ∈ *nodes A*. ⋃ *a* ∈ *alphabet A*. *f* ‘ {*p*} × {*a*} × *f* ‘ {*transition A a p*}

definition *trans-algo* **where**
```
trans-algo N L S f ≡
  FOREACH N (λ p T. do {
    ASSERT (p ∈ N);
    FOREACH L (λ a T. do {
      ASSERT (a ∈ L);
      let q = S a p;
      ASSERT ((f p, a, f q) ∉ T);
      RETURN (insert (f p, a, f q) T) }
    ) T }
  ) {}
```

lemma *trans-algo-refine*:
assumes *finite* (*nodes A*) *finite* (*alphabet A*) *inj-on f* (*nodes A*)
assumes *N* = *nodes A* *L* = *alphabet A* *S* = *transition A*
shows (*trans-algo N L S f*, *SPEC* (*HOL.eq* (*trans-spec A f*))) ∈ ⟨*Id*⟩ *nres-rel*
⟨*proof*⟩

definition *to-draei* :: (*'state*, *'label*) *dra* ⇒ (*'state*, *'label*) *dra*

where *to-draei* ≡ *id*

schematic-goal *to-draei-impl*:
fixes S :: (*'statei* × *'state*) *set*
assumes [*simp*]: *finite* (*nodes A*)
assumes [*autoref-ga-rules*]: *is-bounded-hashcode S seq bhc*
assumes [*autoref-ga-rules*]: *is-valid-def-hm-size TYPE*(*'statei*) *hms*
assumes [*autoref-rules*]: (*seq*, *HOL.eq*) ∈ $S \to S \to$ *bool-rel*
assumes [*autoref-rules*]: (*Ai*, *A*) ∈ ⟨*L*, *S*⟩ *drai-dra-rel*
shows (*?f* :: *?'a*, *do* {
let N = *nodes A*;
f ← *op-set-enumerate N*;
ASSERT (*dom f* = *N*);
ASSERT (*f* (*initial A*) ≠ *None*);
ASSERT (∀ *a* ∈ *alphabet A*. ∀ *p* ∈ *dom f*. *f* (*transition A a p*) ≠ *None*);
T ← *trans-algo N* (*alphabet A*) (*transition A*) (λ *x*. *the* (*f x*));
RETURN (*drae* (*alphabet A*) ((λ *x*. *the* (*f x*)) (*initial A*)) *T*
(*map* (λ (*P*, *Q*). ((λ *x*. *the* (*f x*)) ‘ {*p* ∈ *N*. *P p*}, (λ *x*. *the* (*f x*)) ‘ {*p* ∈ *N*. *Q p*})) (*condition A*)))
}) ∈ *?R*
⟨*proof*⟩
concrete-definition *to-draei-impl* **uses** *to-draei-impl*
lemma *to-draei-impl-refine''*:
fixes S :: (*'statei* × *'state*) *set*
assumes *finite* (*nodes A*)
assumes *is-bounded-hashcode S seq bhc*
assumes *is-valid-def-hm-size TYPE*(*'statei*) *hms*
assumes (*seq*, *HOL.eq*) ∈ $S \to S \to$ *bool-rel*
assumes (*Ai*, *A*) ∈ ⟨*L*, *S*⟩ *drai-dra-rel*
shows (*RETURN* (*to-draei-impl seq bhc hms Ai*), *do* {
f ← *op-set-enumerate* (*nodes A*);
RETURN (*drae-image* (*the* ∘ *f*) (*dra-drae A*))
}) ∈ ⟨⟨*L*, *nat-rel*⟩ *draei-drae-rel*⟩ *nres-rel*
⟨*proof*⟩

context
fixes *Ai A*
fixes *seq bhc hms*
fixes S :: (*'statei* × *'state*) *set*
assumes *a*: *finite* (*nodes A*)
assumes *b*: *is-bounded-hashcode S seq bhc*
assumes *c*: *is-valid-def-hm-size TYPE*(*'statei*) *hms*
assumes *d*: (*seq*, *HOL.eq*) ∈ $S \to S \to$ *bool-rel*
assumes *e*: (*Ai*, *A*) ∈ ⟨*Id*, *S*⟩ *drai-dra-rel*
begin

definition *f'* **where** *f'* ≡ *SOME f'*.

$(to\text{-}draei\text{-}impl\ seq\ bhc\ hms\ Ai,\ drae\text{-}image\ (the \circ f')\ (dra\text{-}drae\ A)) \in \langle Id,\ nat\text{-}rel\rangle\ draei\text{-}drae\text{-}rel \wedge$
$dom\ f' = nodes\ A \wedge inj\text{-}on\ f'\ (nodes\ A)$

lemma *1*: $\exists\ f'.\ (to\text{-}draei\text{-}impl\ seq\ bhc\ hms\ Ai,\ drae\text{-}image\ (the \circ f')\ (dra\text{-}drae\ A)) \in$
$\langle Id,\ nat\text{-}rel\rangle\ draei\text{-}drae\text{-}rel \wedge dom\ f' = nodes\ A \wedge inj\text{-}on\ f'\ (nodes\ A)$
⟨*proof*⟩

lemma f'*-refine*: $(to\text{-}draei\text{-}impl\ seq\ bhc\ hms\ Ai,\ drae\text{-}image\ (the \circ f')\ (dra\text{-}drae\ A)) \in$
$\langle Id,\ nat\text{-}rel\rangle\ draei\text{-}drae\text{-}rel$ ⟨*proof*⟩
lemma f'*-dom*: $dom\ f' = nodes\ A$ ⟨*proof*⟩
lemma f'*-inj*: $inj\text{-}on\ f'\ (nodes\ A)$ ⟨*proof*⟩

definition f **where** $f \equiv the \circ f'$
definition g **where** $g = inv\text{-}into\ (nodes\ A)\ f$
lemma *inj-f*[*intro*!, *simp*]: $inj\text{-}on\ f\ (nodes\ A)$
⟨*proof*⟩
lemma *inj-g*[*intro*!, *simp*]: $inj\text{-}on\ g\ (f\ `\ nodes\ A)$
⟨*proof*⟩

definition *rel* **where** $rel \equiv \{(f\ p,\ p)\ |p.\ p \in nodes\ A\}$
lemma *rel-alt-def*: $rel = (br\ f\ (\lambda\ p.\ p \in nodes\ A))^{-1}$
⟨*proof*⟩
lemma *rel-inv-def*: $rel = br\ g\ (\lambda\ k.\ k \in f\ `\ nodes\ A)$
⟨*proof*⟩
lemma *rel-domain*[*simp*]: $Domain\ rel = f\ `\ nodes\ A$ ⟨*proof*⟩
lemma *rel-range*[*simp*]: $Range\ rel = nodes\ A$ ⟨*proof*⟩
lemma [*intro*!, *simp*]: *bijective rel* ⟨*proof*⟩
lemma [*simp*]: $Id\text{-}on\ (f\ `\ nodes\ A)\ O\ rel = rel$ ⟨*proof*⟩
lemma [*simp*]: $rel\ O\ Id\text{-}on\ (nodes\ A) = rel$ ⟨*proof*⟩

lemma [*param*]: $(f,\ f) \in Id\text{-}on\ (Range\ rel) \to Id\text{-}on\ (Domain\ rel)$ ⟨*proof*⟩
lemma [*param*]: $(g,\ g) \in Id\text{-}on\ (Domain\ rel) \to Id\text{-}on\ (Range\ rel)$ ⟨*proof*⟩
lemma [*param*]: $(id,\ f) \in rel \to Id\text{-}on\ (Domain\ rel)$ ⟨*proof*⟩
lemma [*param*]: $(f,\ id) \in Id\text{-}on\ (Range\ rel) \to rel$ ⟨*proof*⟩
lemma [*param*]: $(id,\ g) \in Id\text{-}on\ (Domain\ rel) \to rel$ ⟨*proof*⟩
lemma [*param*]: $(g,\ id) \in rel \to Id\text{-}on\ (Range\ rel)$ ⟨*proof*⟩

lemma *to-draei-impl-refine'*:
$(to\text{-}draei\text{-}impl\ seq\ bhc\ hms\ Ai,\ to\text{-}draei\ A) \in \langle Id\text{-}on\ (alphabet\ A),\ rel\rangle\ draei\text{-}dra\text{-}rel$
⟨*proof*⟩

end

context
begin

interpretation *autoref-syn* ⟨*proof*⟩

lemma *to-draei-impl-refine*[*autoref-rules*]:
fixes *S* :: (′*statei* × ′*state*) *set*
assumes *SIDE-PRECOND* (*finite* (*nodes A*))
assumes *SIDE-GEN-ALGO* (*is-bounded-hashcode S seq bhc*)
assumes *SIDE-GEN-ALGO* (*is-valid-def-hm-size TYPE*(′*statei*) *hms*)
assumes *GEN-OP seq HOL.eq* (*S* → *S* → *bool-rel*)
assumes (*Ai*, *A*) ∈ ⟨*Id*, *S*⟩ *drai-dra-rel*
shows (*to-draei-impl seq bhc hms Ai*,
(*OP to-draei* ::: ⟨*Id*, *S*⟩ *drai-dra-rel* →
⟨*Id-on* (*alphabet A*), *rel Ai A seq bhc hms*⟩ *draei-dra-rel*) $ *A*) ∈
⟨*Id-on* (*alphabet A*), *rel Ai A seq bhc hms*⟩ *draei-dra-rel*
⟨*proof*⟩

end

end

35 Nondeterministic Büchi Automata

theory *NBA*
imports ../*Nondeterministic*
begin

datatype (′*label*, ′*state*) *nba* = *nba*
(*alphabet*: ′*label set*)
(*initial*: ′*state set*)
(*transition*: ′*label* ⇒ ′*state* ⇒ ′*state set*)
(*accepting*: ′*state pred*)

global-interpretation *nba*: *automaton nba alphabet initial transition accepting*
defines *path* = *nba.path* **and** *run* = *nba.run* **and** *reachable* = *nba.reachable* **and** *nodes* = *nba.nodes*
⟨*proof*⟩
global-interpretation *nba*: *automaton-run nba alphabet initial transition accepting* λ *P w r p. infs P* (*p* ## *r*)
defines *language* = *nba.language*
⟨*proof*⟩

abbreviation *target* **where** *target* ≡ *nba.target*
abbreviation *states* **where** *states* ≡ *nba.states*
abbreviation *trace* **where** *trace* ≡ *nba.trace*
abbreviation *successors* **where** *successors* ≡ *nba.successors TYPE*(′*label*)

instantiation *nba* :: (*type*, *type*) *order*
begin

definition *less-eq-nba* :: (′*a*, ′*b*) *nba* ⇒ (′*a*, ′*b*) *nba* ⇒ *bool* **where**

$A \le B \equiv$ *alphabet* $A \le$ *alphabet* $B \wedge$ *initial* $A \le$ *initial* $B \wedge$
transition $A \le$ *transition* $B \wedge$ *accepting* $A \le$ *accepting* B
definition *less-nba* :: $('a, 'b)$ *nba* $\Rightarrow$ $('a, 'b)$ *nba* $\Rightarrow$ *bool* **where**
less-nba $A\ B \equiv A \le B \wedge A \ne B$

instance ⟨*proof*⟩

end

lemma *nodes-mono*: *mono nodes*
⟨*proof*⟩

lemma *language-mono*: *mono language*
⟨*proof*⟩

lemma *simulation-language*:
assumes *alphabet* $A \subseteq$ *alphabet* B
assumes $\bigwedge p.\ p \in$ *initial* $A \Longrightarrow \exists\ q \in$ *initial* $B.\ (p, q) \in R$
assumes $\bigwedge a\ p\ p'\ q.\ p' \in$ *transition* $A\ a\ p \Longrightarrow (p, q) \in R \Longrightarrow \exists\ q' \in$ *transition* $B\ a\ q.\ (p', q') \in R$
assumes $\bigwedge p\ q.\ (p, q) \in R \Longrightarrow$ *accepting* $A\ p \Longrightarrow$ *accepting* $B\ q$
shows *language* $A \subseteq$ *language* B
⟨*proof*⟩

end

36 Nondeterministic Generalized Büchi Automata

theory *NGBA*
imports ../*Nondeterministic*
begin

datatype $('label, 'state)$ *ngba* = *ngba*
(*alphabet*: *'label set*)
(*initial*: *'state set*)
(*transition*: *'label* $\Rightarrow$ *'state* $\Rightarrow$ *'state set*)
(*accepting*: *'state pred gen*)

global-interpretation *ngba*: *automaton ngba alphabet initial transition accepting*
defines *path* = *ngba.path* **and** *run* = *ngba.run* **and** *reachable* = *ngba.reachable* **and** *nodes* = *ngba.nodes*
⟨*proof*⟩
global-interpretation *ngba*: *automaton-run ngba alphabet initial transition accepting* $\lambda\ P\ w\ r\ p.$ *gen infs* $P\ (p\ \#\#\ r)$
defines *language* = *ngba.language*
⟨*proof*⟩

abbreviation *target* **where** *target* $\equiv$ *ngba.target*
abbreviation *states* **where** *states* $\equiv$ *ngba.states*

abbreviation *trace* **where** *trace* ≡ *ngba.trace*
abbreviation *successors* **where** *successors* ≡ *ngba.successors TYPE('label)*

end

37 Nondeterministic Büchi Automata Combinations

theory *NBA-Combine*
imports *NBA NGBA*
begin

global-interpretation *degeneralization*: *automaton-degeneralization-run*
ngba ngba.alphabet ngba.initial ngba.transition ngba.accepting λ *P w r p. gen infs P* (*p* ## *r*)
nba nba.alphabet nba.initial nba.transition nba.accepting λ *P w r p. infs P* (*p* ## *r*)
fst id
defines *degeneralize* = *degeneralization.degeneralize*
⟨*proof*⟩

lemmas *degeneralize-language*[*simp*] = *degeneralization.degeneralize-language*[*folded NBA.language-def*]
lemmas *degeneralize-nodes-finite*[*iff*] = *degeneralization.degeneralize-nodes-finite*[*folded NBA.nodes-def*]

global-interpretation *intersection*: *automaton-intersection-run*
nba nba.alphabet nba.initial nba.transition nba.accepting λ *P w r p. infs P* (*p* ## *r*)
nba nba.alphabet nba.initial nba.transition nba.accepting λ *P w r p. infs P* (*p* ## *r*)
ngba ngba.alphabet ngba.initial ngba.transition ngba.accepting λ *P w r p. gen infs P* (*p* ## *r*)
$\lambda\ c_1\ c_2.\ [c_1 \circ fst,\ c_2 \circ snd]$
defines *intersect'* = *intersection.product*
⟨*proof*⟩

lemmas *intersect'-language*[*simp*] = *intersection.product-language*[*folded NGBA.language-def*]
lemmas *intersect'-nodes-finite*[*intro*] = *intersection.product-nodes-finite*[*folded NGBA.nodes-def*]

global-interpretation *union*: *automaton-union-run*
nba nba.alphabet nba.initial nba.transition nba.accepting λ *P w r p. infs P* (*p* ## *r*)
nba nba.alphabet nba.initial nba.transition nba.accepting λ *P w r p. infs P* (*p* ## *r*)
nba nba.alphabet nba.initial nba.transition nba.accepting λ *P w r p. infs P* (*p* ## *r*)

case-sum
defines *union* = *union.sum*
⟨*proof*⟩

lemmas *union-language* = *union.sum-language*
lemmas *union-nodes-finite* = *union.sum-nodes-finite*

global-interpretation *intersection-list*: *automaton-intersection-list-run*
nba nba.alphabet nba.initial nba.transition nba.accepting λ *P w r p. infs P* (*p* ## *r*)
ngba ngba.alphabet ngba.initial ngba.transition ngba.accepting λ *P w r p. gen infs P* (*p* ## *r*)
λ *cs. map* (λ *k ps.* (*cs* ! *k*) (*ps* ! *k*)) [*0* ..< *length cs*]
defines *intersect-list′* = *intersection-list.product*
⟨*proof*⟩

lemmas *intersect-list′-language*[*simp*] = *intersection-list.product-language*[*folded NGBA.language-def*]
lemmas *intersect-list′-nodes-finite*[*intro*] = *intersection-list.product-nodes-finite*[*folded NGBA.nodes-def*]

global-interpretation *union-list*: *automaton-union-list-run*
nba nba.alphabet nba.initial nba.transition nba.accepting λ *P w r p. infs P* (*p* ## *r*)
nba nba.alphabet nba.initial nba.transition nba.accepting λ *P w r p. infs P* (*p* ## *r*)
λ *cs* (*k*, *p*). (*cs* ! *k*) *p*
defines *union-list* = *union-list.sum*
⟨*proof*⟩

lemmas *union-list-language* = *union-list.sum-language*
lemmas *union-list-nodes-finite* = *union-list.sum-nodes-finite*

abbreviation *intersect* **where** *intersect A B* ≡ *degeneralize* (*intersect′ A B*)

lemma *intersect-language*[*simp*]: *NBA.language* (*intersect A B*) = *NBA.language A* ∩ *NBA.language B*
⟨*proof*⟩
lemma *intersect-nodes-finite*[*intro*]:
assumes *finite* (*NBA.nodes A*) *finite* (*NBA.nodes B*)
shows *finite* (*NBA.nodes* (*intersect A B*))
⟨*proof*⟩

abbreviation *intersect-list* **where** *intersect-list AA* ≡ *degeneralize* (*intersect-list′ AA*)

lemma *intersect-list-language*[*simp*]: *NBA.language* (*intersect-list AA*) = ⋂ (*NBA.language* ‘ *set AA*)
⟨*proof*⟩

lemma *intersect-list-nodes-finite*[*intro*]:
assumes *list-all* (*finite* ∘ *NBA.nodes*) *AA*
shows *finite* (*NBA.nodes* (*intersect-list AA*))
⟨*proof*⟩

end

38 Connecting Nondeterministic Büchi Automata to CAVA Automata Structures

theory *NBA-Graphs*
imports
NBA
CAVA-Automata.Automata-Impl
begin

no-notation *build* (**infixr** *## 65*)

38.1 Regular Graphs

definition *nba-g* :: (*'label*, *'state*) *nba* ⇒ *'state graph-rec* **where**
nba-g A ≡ ⦇ *g-V* = *UNIV*, *g-E* = *E-of-succ* (*successors A*), *g-V0* = *initial A* ⦈

lemma *nba-g-graph*[*simp*]: *graph* (*nba-g A*) ⟨*proof*⟩

lemma *nba-g-V0*: *g-V0* (*nba-g A*) = *initial A* ⟨*proof*⟩
lemma *nba-g-E-rtrancl*: $(g\text{-}E\ (nba\text{-}g\ A))^* = \{(p,\ q).\ q \in reachable\ A\ p\}$
⟨*proof*⟩

lemma *nba-g-rtrancl-path*: $(g\text{-}E\ (nba\text{-}g\ A))^* = \{(p,\ target\ r\ p)\ |r\ p.\ NBA.path\ A\ r\ p\}$
⟨*proof*⟩
lemma *nba-g-trancl-path*: $(g\text{-}E\ (nba\text{-}g\ A))^+ = \{(p,\ target\ r\ p)\ |r\ p.\ NBA.path\ A\ r\ p \wedge r \neq []\}$
⟨*proof*⟩

lemma *nba-g-ipath-run*:
assumes *ipath* (*g-E* (*nba-g A*)) *r*
obtains *w*
where *run A* (*w* ||| *smap* (*r* ∘ *Suc*) *nats*) (*r 0*)
⟨*proof*⟩
lemma *nba-g-run-ipath*:
assumes *run A* (*w* ||| *r*) *p*
shows *ipath* (*g-E* (*nba-g A*)) (*snth* (*p ## r*))
⟨*proof*⟩

38.2 Indexed Generalized Büchi Graphs

definition *nba-igbg* :: (*'label*, *'state*) *nba* ⇒ *'state igb-graph-rec* **where**

nba-igbg A $\equiv$ *graph-rec.extend* (*nba-g A*)
$(\!|$ *igbg-num-acc* = *1*, *igbg-acc* = λ *p*. *if accepting A p then* {*0*} *else* {} $|\!)$

lemma *acc-run-language*:
assumes *igb-graph* (*nba-igbg A*)
shows *Ex* (*igb-graph.is-acc-run* (*nba-igbg A*)) $\longleftrightarrow$ *language A* $\neq$ {}
$\langle$*proof*$\rangle$

end

39 Relations on Nondeterministic Büchi Automata

theory *NBA-Refine*
imports
NBA
../../Transition-Systems/Transition-System-Refine
begin

definition *nba-rel* :: ($'label_1$ $\times$ $'label_2$) *set* $\Rightarrow$ ($'state_1$ $\times$ $'state_2$) *set* $\Rightarrow$
(($'label_1$, $'state_1$) *nba* $\times$ ($'label_2$, $'state_2$) *nba*) *set* **where**
[*to-relAPP*]: *nba-rel L S* $\equiv$ {(A_1, A_2).
(*alphabet* A_1, *alphabet* A_2) $\in$ $\langle L\rangle$ *set-rel* $\wedge$
(*initial* A_1, *initial* A_2) $\in$ $\langle S\rangle$ *set-rel* $\wedge$
(*transition* A_1, *transition* A_2) $\in$ $L \to S \to \langle S\rangle$ *set-rel* $\wedge$
(*accepting* A_1, *accepting* A_2) $\in$ $S \to$ *bool-rel*}

lemma *nba-param*[*param*]:
(*nba*, *nba*) $\in$ $\langle L\rangle$ *set-rel* $\to$ $\langle S\rangle$ *set-rel* $\to$ ($L \to S \to \langle S\rangle$ *set-rel*) $\to$ ($S \to$ *bool-rel*) $\to$
$\langle L, S\rangle$ *nba-rel*
(*alphabet*, *alphabet*) $\in$ $\langle L, S\rangle$ *nba-rel* $\to$ $\langle L\rangle$ *set-rel*
(*initial*, *initial*) $\in$ $\langle L, S\rangle$ *nba-rel* $\to$ $\langle S\rangle$ *set-rel*
(*transition*, *transition*) $\in$ $\langle L, S\rangle$ *nba-rel* $\to L \to S \to \langle S\rangle$ *set-rel*
(*accepting*, *accepting*) $\in$ $\langle L, S\rangle$ *nba-rel* $\to S \to$ *bool-rel*
$\langle$*proof*$\rangle$

lemma *nba-rel-id*[*simp*]: $\langle Id, Id\rangle$ *nba-rel* = *Id* $\langle$*proof*$\rangle$
lemma *nba-rel-comp*[*trans*]:
assumes [*param*]: (*A*, *B*) $\in$ $\langle L_1, S_1\rangle$ *nba-rel* (*B*, *C*) $\in$ $\langle L_2, S_2\rangle$ *nba-rel*
shows (*A*, *C*) $\in$ $\langle L_1\ O\ L_2, S_1\ O\ S_2\rangle$ *nba-rel*
$\langle$*proof*$\rangle$
lemma *nba-rel-converse*[*simp*]: ($\langle L, S\rangle$ *nba-rel*)$^{-1}$ = $\langle L^{-1}, S^{-1}\rangle$ *nba-rel*
$\langle$*proof*$\rangle$

lemma *nba-rel-eq*: (*A*, *A*) $\in$ $\langle$*Id-on* (*alphabet A*), *Id-on* (*nodes A*)$\rangle$ *nba-rel*
$\langle$*proof*$\rangle$

lemma *enableds-param*[*param*]: (*nba.enableds*, *nba.enableds*) $\in$ $\langle L, S\rangle$ *nba-rel* $\to$ $S \to \langle L \times_r S\rangle$ *set-rel*

$\langle proof \rangle$

lemma *paths-param*[*param*]: (*nba.paths*, *nba.paths*) $\in \langle L, S\rangle$ *nba-rel* $\to S \to \langle\langle L \times_r S\rangle$ *list-rel*$\rangle$ *set-rel*

$\langle proof \rangle$

lemma *runs-param*[*param*]: (*nba.runs*, *nba.runs*) $\in \langle L, S\rangle$ *nba-rel* $\to S \to \langle\langle L \times_r S\rangle$ *stream-rel*$\rangle$ *set-rel*

$\langle proof \rangle$

lemma *reachable-param*[*param*]: (*reachable*, *reachable*) $\in \langle L, S\rangle$ *nba-rel* $\to S \to \langle S\rangle$ *set-rel*

$\langle proof \rangle$

lemma *nodes-param*[*param*]: (*nodes*, *nodes*) $\in \langle L, S\rangle$ *nba-rel* $\to \langle S\rangle$ *set-rel*

$\langle proof \rangle$

lemma *language-param*[*param*]: (*language*, *language*) $\in \langle L, S\rangle$ *nba-rel* $\to \langle\langle L\rangle$ *stream-rel*$\rangle$ *set-rel*

$\langle proof \rangle$

end

40 Implementation of Nondeterministic Büchi Automata

theory *NBA-Implement*
imports
NBA-Refine
../../Basic/Implement
begin

consts *i-nba-scheme* :: *interface* $\Rightarrow$ *interface* $\Rightarrow$ *interface*

context
begin

interpretation *autoref-syn* $\langle proof \rangle$

lemma *nba-scheme-itype*[*autoref-itype*]:
nba $::_i \langle L\rangle_i$ *i-set* $\to_i \langle S\rangle_i$ *i-set* $\to_i$ ($L \to_i S \to_i \langle S\rangle_i$ *i-set*) $\to_i \langle S\rangle_i$ *i-set* $\to_i$ $\langle L, S\rangle_i$ *i-nba-scheme*
alphabet $::_i \langle L, S\rangle_i$ *i-nba-scheme* $\to_i \langle L\rangle_i$ *i-set*
initial $::_i \langle L, S\rangle_i$ *i-nba-scheme* $\to_i \langle S\rangle_i$ *i-set*
transition $::_i \langle L, S\rangle_i$ *i-nba-scheme* $\to_i L \to_i S \to_i \langle S\rangle_i$ *i-set*
accepting $::_i \langle L, S\rangle_i$ *i-nba-scheme* $\to_i \langle S\rangle_i$ *i-set*
$\langle proof \rangle$

end

datatype (*'label*, *'state*) *nbai* = *nbai*

$(alphabeti: 'label\ list)$
$(initiali: 'state\ list)$
$(transitioni: 'label \Rightarrow 'state \Rightarrow 'state\ list)$
$(acceptingi: 'state \Rightarrow bool)$

definition *nbai-rel* :: $('label_1 \times 'label_2)\ set \Rightarrow ('state_1 \times 'state_2)\ set \Rightarrow$
$(('label_1, 'state_1)\ nbai \times ('label_2, 'state_2)\ nbai)\ set$ **where**
$[to\text{-}relAPP]$: $nbai\text{-}rel\ L\ S \equiv \{(A_1, A_2).$
$(alphabeti\ A_1, alphabeti\ A_2) \in \langle L \rangle\ list\text{-}rel\ \wedge$
$(initiali\ A_1, initiali\ A_2) \in \langle S \rangle\ list\text{-}rel\ \wedge$
$(transitioni\ A_1, transitioni\ A_2) \in L \rightarrow S \rightarrow \langle S \rangle\ list\text{-}rel\ \wedge$
$(acceptingi\ A_1, acceptingi\ A_2) \in S \rightarrow bool\text{-}rel\}$

lemma *nbai-param*[*param*, *autoref-rules*]:
$(nbai, nbai) \in \langle L \rangle\ list\text{-}rel \rightarrow \langle S \rangle\ list\text{-}rel \rightarrow (L \rightarrow S \rightarrow \langle S \rangle\ list\text{-}rel) \rightarrow$
$(S \rightarrow bool\text{-}rel) \rightarrow \langle L, S \rangle\ nbai\text{-}rel$
$(alphabeti, alphabeti) \in \langle L, S \rangle\ nbai\text{-}rel \rightarrow \langle L \rangle\ list\text{-}rel$
$(initiali, initiali) \in \langle L, S \rangle\ nbai\text{-}rel \rightarrow \langle S \rangle\ list\text{-}rel$
$(transitioni, transitioni) \in \langle L, S \rangle\ nbai\text{-}rel \rightarrow L \rightarrow S \rightarrow \langle S \rangle\ list\text{-}rel$
$(acceptingi, acceptingi) \in \langle L, S \rangle\ nbai\text{-}rel \rightarrow (S \rightarrow bool\text{-}rel)$
$\langle proof \rangle$

definition *nbai-nba-rel* :: $('label_1 \times 'label_2)\ set \Rightarrow ('state_1 \times 'state_2)\ set \Rightarrow$
$(('label_1, 'state_1)\ nbai \times ('label_2, 'state_2)\ nba)\ set$ **where**
$[to\text{-}relAPP]$: $nbai\text{-}nba\text{-}rel\ L\ S \equiv \{(A_1, A_2).$
$(alphabeti\ A_1, alphabet\ A_2) \in \langle L \rangle\ list\text{-}set\text{-}rel\ \wedge$
$(initiali\ A_1, initial\ A_2) \in \langle S \rangle\ list\text{-}set\text{-}rel\ \wedge$
$(transitioni\ A_1, transition\ A_2) \in L \rightarrow S \rightarrow \langle S \rangle\ list\text{-}set\text{-}rel\ \wedge$
$(acceptingi\ A_1, accepting\ A_2) \in S \rightarrow bool\text{-}rel\}$

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of nbai-nba-rel i-nba-scheme*]

lemma *nbai-nba-param*[*param*, *autoref-rules*]:
$(nbai, nba) \in \langle L \rangle\ list\text{-}set\text{-}rel \rightarrow \langle S \rangle\ list\text{-}set\text{-}rel \rightarrow (L \rightarrow S \rightarrow \langle S \rangle\ list\text{-}set\text{-}rel) \rightarrow$
$(S \rightarrow bool\text{-}rel) \rightarrow \langle L, S \rangle\ nbai\text{-}nba\text{-}rel$
$(alphabeti, alphabet) \in \langle L, S \rangle\ nbai\text{-}nba\text{-}rel \rightarrow \langle L \rangle\ list\text{-}set\text{-}rel$
$(initiali, initial) \in \langle L, S \rangle\ nbai\text{-}nba\text{-}rel \rightarrow \langle S \rangle\ list\text{-}set\text{-}rel$
$(transitioni, transition) \in \langle L, S \rangle\ nbai\text{-}nba\text{-}rel \rightarrow L \rightarrow S \rightarrow \langle S \rangle\ list\text{-}set\text{-}rel$
$(acceptingi, accepting) \in \langle L, S \rangle\ nbai\text{-}nba\text{-}rel \rightarrow S \rightarrow bool\text{-}rel$
$\langle proof \rangle$

definition *nbai-nba* :: $('label, 'state)\ nbai \Rightarrow ('label, 'state)\ nba$ **where**
$nbai\text{-}nba\ A \equiv nba\ (set\ (alphabeti\ A))\ (set\ (initiali\ A))\ (\lambda\ a\ p.\ set\ (transitioni\ A\ a\ p))\ (acceptingi\ A)$
definition *nbai-invar* :: $('label, 'state)\ nbai \Rightarrow bool$ **where**
$nbai\text{-}invar\ A \equiv distinct\ (alphabeti\ A) \wedge distinct\ (initiali\ A) \wedge (\forall\ a\ p.\ distinct\ (transitioni\ A\ a\ p))$

lemma *nbai-nba-id-param*[*param*]: (*nbai-nba*, *id*) ∈ ⟨*L*, *S*⟩ *nbai-nba-rel* → ⟨*L*, *S*⟩ *nba-rel*
⟨*proof*⟩

lemma *nbai-nba-br*: ⟨*Id*, *Id*⟩ *nbai-nba-rel* = *br nbai-nba nbai-invar*
⟨*proof*⟩

end

41 Algorithms on Nondeterministic Büchi Automata

theory *NBA-Algorithms*
imports
NBA-Graphs
NBA-Implement
DFS-Framework.Reachable-Nodes
Gabow-SCC.Gabow-GBG-Code
begin

41.1 Miscellaneous Amendments

lemma (**in** *igb-fr-graph*) *acc-run-lasso-prpl*: *Ex is-acc-run* ⟹ *Ex is-lasso-prpl*
⟨*proof*⟩
lemma (**in** *igb-fr-graph*) *lasso-prpl-acc-run-iff*: *Ex is-lasso-prpl* ⟷ *Ex is-acc-run*
⟨*proof*⟩

lemma [*autoref-rel-intf*]: *REL-INTF igbg-impl-rel-ext i-igbg* ⟨*proof*⟩

41.2 Operations

definition *op-language-empty* **where** [*simp*]: *op-language-empty A* ≡ *language A* = {}

lemmas [*autoref-op-pat*] = *op-language-empty-def*[*symmetric*]

41.3 Implementations

context
begin

interpretation *autoref-syn* ⟨*proof*⟩

lemma *nba-g-ahs*: *nba-g A* = (| *g-V* = *UNIV*, *g-E* = *E-of-succ* (λ *p*. *CAST* ((⋃ *a* ∈ *alphabet A*. *transition A a p* ::: ⟨*S*⟩ *list-set-rel*) ::: ⟨*S*⟩ *ahs-rel bhc*)), *g-V0* = *initial A* |)
⟨*proof*⟩

schematic-goal *nbai-gi*:
notes [*autoref-ga-rules*] = *map2set-to-list*

fixes S :: ('*statei* × '*state*) *set*
assumes [*autoref-ga-rules*]: *is-bounded-hashcode S seq bhc*
assumes [*autoref-ga-rules*]: *is-valid-def-hm-size TYPE*('*statei*) *hms*
assumes [*autoref-rules*]: (*seq*, *HOL.eq*) ∈ $S \to S \to$ *bool-rel*
assumes [*autoref-rules*]: (*Ai*, *A*) ∈ ⟨*L*, *S*⟩ *nbai-nba-rel*
shows (*?f* :: *?'a*, *RETURN* (*nba-g A*)) ∈ *?A*
⟨*proof*⟩
concrete-definition *nbai-gi* **uses** *nbai-gi*
lemma *nbai-gi-refine*[*autoref-rules*]:
fixes S :: ('*statei* × '*state*) *set*
assumes *SIDE-GEN-ALGO* (*is-bounded-hashcode S seq bhc*)
assumes *SIDE-GEN-ALGO* (*is-valid-def-hm-size TYPE*('*statei*) *hms*)
assumes *GEN-OP seq HOL.eq* ($S \to S \to$ *bool-rel*)
shows (*NBA-Algorithms.nbai-gi seq bhc hms*, *nba-g*) ∈
⟨*L*, *S*⟩ *nbai-nba-rel* → ⟨*unit-rel*, *S*⟩ *g-impl-rel-ext*
⟨*proof*⟩

schematic-goal *nba-nodes*:
fixes S :: ('*statei* × '*state*) *set*
assumes [*simp*]: *finite* ((*g-E* (*nba-g A*))* ‘‘ *g-V0* (*nba-g A*))
assumes [*autoref-ga-rules*]: *is-bounded-hashcode S seq bhc*
assumes [*autoref-ga-rules*]: *is-valid-def-hm-size TYPE*('*statei*) *hms*
assumes [*autoref-rules*]: (*seq*, *HOL.eq*) ∈ $S \to S \to$ *bool-rel*
assumes [*autoref-rules*]: (*Ai*, *A*) ∈ ⟨*L*, *S*⟩ *nbai-nba-rel*
shows (*?f* :: *?'a*, *op-reachable* (*nba-g A*)) ∈ *?R* ⟨*proof*⟩
concrete-definition *nba-nodes* **uses** *nba-nodes*
lemma *nba-nodes-refine*[*autoref-rules*]:
fixes S :: ('*statei* × '*state*) *set*
assumes *SIDE-PRECOND* (*finite* (*nodes A*))
assumes *SIDE-GEN-ALGO* (*is-bounded-hashcode S seq bhc*)
assumes *SIDE-GEN-ALGO* (*is-valid-def-hm-size TYPE*('*statei*) *hms*)
assumes *GEN-OP seq HOL.eq* ($S \to S \to$ *bool-rel*)
assumes (*Ai*, *A*) ∈ ⟨*L*, *S*⟩ *nbai-nba-rel*
shows (*NBA-Algorithms.nba-nodes seq bhc hms Ai*,
(*OP nodes* ::: ⟨*L*, *S*⟩ *nbai-nba-rel* → ⟨*S*⟩ *ahs-rel bhc*) $ *A*) ∈ ⟨*S*⟩ *ahs-rel bhc*
⟨*proof*⟩

lemma *nba-igbg-ahs*: *nba-igbg A* = ⦇ *g-V* = *UNIV*, *g-E* = *E-of-succ* (λ *p*. *CAST*
((⋃ *a* ∈ *alphabet A*. *transition A a p* ::: ⟨*S*⟩ *list-set-rel*) ::: ⟨*S*⟩ *ahs-rel bhc*)), *g-V0* = *initial A*,
igbg-num-acc = *1*, *igbg-acc* = λ *p*. *if accepting A p then* {*0*} *else* {} ⦈
⟨*proof*⟩

schematic-goal *nbai-igbgi*:
notes [*autoref-ga-rules*] = *map2set-to-list*
fixes S :: ('*statei* × '*state*) *set*
assumes [*autoref-ga-rules*]: *is-bounded-hashcode S seq bhc*
assumes [*autoref-ga-rules*]: *is-valid-def-hm-size TYPE*('*statei*) *hms*

assumes [*autoref-rules*]: (*seq*, *HOL.eq*) ∈ *S* → *S* → *bool-rel*
assumes [*autoref-rules*]: (*Ai*, *A*) ∈ ⟨*L*, *S*⟩ *nbai-nba-rel*
shows (*?f* :: *?'a*, *RETURN* (*nba-igbg A*)) ∈ *?A*
⟨*proof*⟩
concrete-definition *nbai-igbgi* **uses** *nbai-igbgi*
lemma *nbai-igbgi-refine*[*autoref-rules*]:
fixes *S* :: (*'statei* × *'state*) *set*
assumes *SIDE-GEN-ALGO* (*is-bounded-hashcode S seq bhc*)
assumes *SIDE-GEN-ALGO* (*is-valid-def-hm-size TYPE*(*'statei*) *hms*)
assumes *GEN-OP seq HOL.eq* (*S* → *S* → *bool-rel*)
shows (*NBA-Algorithms.nbai-igbgi seq bhc hms*, *nba-igbg*) ∈
⟨*L*, *S*⟩ *nbai-nba-rel* → *igbg-impl-rel-ext unit-rel S*
⟨*proof*⟩

schematic-goal *nba-language-empty*:
fixes *S* :: (*'statei* × *'state*) *set*
assumes [*simp*]: *igb-fr-graph* (*nba-igbg A*)
assumes [*autoref-ga-rules*]: *is-bounded-hashcode S seq bhs*
assumes [*autoref-ga-rules*]: *is-valid-def-hm-size TYPE*(*'statei*) *hms*
assumes [*autoref-rules*]: (*seq*, *HOL.eq*) ∈ *S* → *S* → *bool-rel*
assumes [*autoref-rules*]: (*Ai*, *A*) ∈ ⟨*L*, *S*⟩ *nbai-nba-rel*
shows (*?f* :: *?'a*, *do* { *r* ← *op-find-lasso-spec* (*nba-igbg A*); *RETURN* (*r* = *None*)}) ∈ *?A*
⟨*proof*⟩
concrete-definition *nba-language-empty* **uses** *nba-language-empty*
lemma *nba-language-empty-refine*[*autoref-rules*]:
fixes *S* :: (*'statei* × *'state*) *set*
assumes *SIDE-PRECOND* (*finite* (*nodes A*))
assumes *SIDE-GEN-ALGO* (*is-bounded-hashcode S seq bhc*)
assumes *SIDE-GEN-ALGO* (*is-valid-def-hm-size TYPE*(*'statei*) *hms*)
assumes *GEN-OP seq HOL.eq* (*S* → *S* → *bool-rel*)
assumes (*Ai*, *A*) ∈ ⟨*L*, *S*⟩ *nbai-nba-rel*
shows (*NBA-Algorithms.nba-language-empty seq bhc hms Ai*,
(*OP op-language-empty* ::: ⟨*L*, *S*⟩ *nbai-nba-rel* → *bool-rel*) $ *A*) ∈ *bool-rel*
⟨*proof*⟩

end

end

42 Explicit Nondeterministic Büchi Automata

theory *NBA-Explicit*
imports *NBA-Algorithms*
begin

datatype (*'label*, *'state*) *nbae* = *nbae*
(*alphabete*: *'label set*)
(*initiale*: *'state set*)

($transitione$: ($'state \times 'label \times 'state$) set)
($acceptinge$: $'state$ set)

definition *nbae-rel* **where**
[*to-relAPP*]: $nbae\text{-}rel\ L\ S \equiv \{(A_1, A_2).$
$(alphabete\ A_1, alphabete\ A_2) \in \langle L\rangle\ set\text{-}rel \wedge$
$(initiale\ A_1, initiale\ A_2) \in \langle S\rangle\ set\text{-}rel \wedge$
$(transitione\ A_1, transitione\ A_2) \in \langle S \times_r L \times_r S\rangle\ set\text{-}rel \wedge$
$(acceptinge\ A_1, acceptinge\ A_2) \in \langle S\rangle\ set\text{-}rel\}$

lemma *nbae-param*[*param*, *autoref-rules*]:
$(nbae, nbae) \in \langle L\rangle\ set\text{-}rel \to \langle S\rangle\ set\text{-}rel \to \langle S \times_r L \times_r S\rangle\ set\text{-}rel \to$
$\langle S\rangle\ set\text{-}rel \to \langle L, S\rangle\ nbae\text{-}rel$
$(alphabete, alphabete) \in \langle L, S\rangle\ nbae\text{-}rel \to \langle L\rangle\ set\text{-}rel$
$(initiale, initiale) \in \langle L, S\rangle\ nbae\text{-}rel \to \langle S\rangle\ set\text{-}rel$
$(transitione, transitione) \in \langle L, S\rangle\ nbae\text{-}rel \to \langle S \times_r L \times_r S\rangle\ set\text{-}rel$
$(acceptinge, acceptinge) \in \langle L, S\rangle\ nbae\text{-}rel \to \langle S\rangle\ set\text{-}rel$
⟨*proof*⟩

lemma *nbae-rel-id*[*simp*]: $\langle Id, Id\rangle\ nbae\text{-}rel = Id$ ⟨*proof*⟩
lemma *nbae-rel-comp*[*simp*]: $\langle L_1\ O\ L_2, S_1\ O\ S_2\rangle\ nbae\text{-}rel = \langle L_1, S_1\rangle\ nbae\text{-}rel\ O$
$\langle L_2, S_2\rangle\ nbae\text{-}rel$
⟨*proof*⟩

consts *i-nbae-scheme* :: *interface* ⇒ *interface* ⇒ *interface*

context
begin

interpretation *autoref-syn* ⟨*proof*⟩

lemma *nbae-scheme-itype*[*autoref-itype*]:
$nbae ::_i \langle L\rangle_i\ i\text{-}set \to_i \langle S\rangle_i\ i\text{-}set \to_i \langle\langle S, \langle L, S\rangle_i\ i\text{-}prod\rangle_i\ i\text{-}prod\rangle_i\ i\text{-}set \to_i \langle S\rangle_i$
$i\text{-}set \to_i$
$\langle L, S\rangle_i\ i\text{-}nbae\text{-}scheme$
$alphabete ::_i \langle L, S\rangle_i\ i\text{-}nbae\text{-}scheme \to_i \langle L\rangle_i\ i\text{-}set$
$initiale ::_i \langle L, S\rangle_i\ i\text{-}nbae\text{-}scheme \to_i \langle S\rangle_i\ i\text{-}set$
$transitione ::_i \langle L, S\rangle_i\ i\text{-}nbae\text{-}scheme \to_i \langle\langle S, \langle L, S\rangle_i\ i\text{-}prod\rangle_i\ i\text{-}prod\rangle_i\ i\text{-}set$
$acceptinge ::_i \langle L, S\rangle_i\ i\text{-}nbae\text{-}scheme \to_i \langle S\rangle_i\ i\text{-}set$
⟨*proof*⟩

end

datatype ($'label$, $'state$) *nbaei* = *nbaei*
($alphabetei$: $'label$ $list$)
($initialei$: $'state$ $list$)
($transitionei$: ($'state \times 'label \times 'state$) $list$)
($acceptingei$: $'state$ $list$)

definition *nbaei-rel* **where**
[*to-relAPP*]: *nbaei-rel L S* ≡ {(A_1, A_2).
(*alphabetei* A_1, *alphabetei* A_2) ∈ ⟨*L*⟩ *list-rel* ∧
(*initialei* A_1, *initialei* A_2) ∈ ⟨*S*⟩ *list-rel* ∧
(*transitionei* A_1, *transitionei* A_2) ∈ ⟨$S \times_r L \times_r S$⟩ *list-rel* ∧
(*acceptingei* A_1, *acceptingei* A_2) ∈ ⟨*S*⟩ *list-rel*}

lemma *nbaei-param*[*param*, *autoref-rules*]:
(*nbaei*, *nbaei*) ∈ ⟨*L*⟩ *list-rel* → ⟨*S*⟩ *list-rel* → ⟨$S \times_r L \times_r S$⟩ *list-rel* →
⟨*S*⟩ *list-rel* → ⟨*L*, *S*⟩ *nbaei-rel*
(*alphabetei*, *alphabetei*) ∈ ⟨*L*, *S*⟩ *nbaei-rel* → ⟨*L*⟩ *list-rel*
(*initialei*, *initialei*) ∈ ⟨*L*, *S*⟩ *nbaei-rel* → ⟨*S*⟩ *list-rel*
(*transitionei*, *transitionei*) ∈ ⟨*L*, *S*⟩ *nbaei-rel* → ⟨$S \times_r L \times_r S$⟩ *list-rel*
(*acceptingei*, *acceptingei*) ∈ ⟨*L*, *S*⟩ *nbaei-rel* → ⟨*S*⟩ *list-rel*
⟨*proof*⟩

definition *nbaei-nbae-rel* **where**
[*to-relAPP*]: *nbaei-nbae-rel L S* ≡ {(A_1, A_2).
(*alphabetei* A_1, *alphabete* A_2) ∈ ⟨*L*⟩ *list-set-rel* ∧
(*initialei* A_1, *initiale* A_2) ∈ ⟨*S*⟩ *list-set-rel* ∧
(*transitionei* A_1, *transitione* A_2) ∈ ⟨$S \times_r L \times_r S$⟩ *list-set-rel* ∧
(*acceptingei* A_1, *acceptinge* A_2) ∈ ⟨*S*⟩ *list-set-rel*}

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of nbaei-nbae-rel i-nbae-scheme*]

lemma *nbaei-nbae-param*[*param*, *autoref-rules*]:
(*nbaei*, *nbae*) ∈ ⟨*L*⟩ *list-set-rel* → ⟨*S*⟩ *list-set-rel* → ⟨$S \times_r L \times_r S$⟩ *list-set-rel*
→
⟨*S*⟩ *list-set-rel* → ⟨*L*, *S*⟩ *nbaei-nbae-rel*
(*alphabetei*, *alphabete*) ∈ ⟨*L*, *S*⟩ *nbaei-nbae-rel* → ⟨*L*⟩ *list-set-rel*
(*initialei*, *initiale*) ∈ ⟨*L*, *S*⟩ *nbaei-nbae-rel* → ⟨*S*⟩ *list-set-rel*
(*transitionei*, *transitione*) ∈ ⟨*L*, *S*⟩ *nbaei-nbae-rel* → ⟨$S \times_r L \times_r S$⟩ *list-set-rel*
(*acceptingei*, *acceptinge*) ∈ ⟨*L*, *S*⟩ *nbaei-nbae-rel* → ⟨*S*⟩ *list-set-rel*
⟨*proof*⟩

definition *nbaei-nbae* **where**
nbaei-nbae A ≡ *nbae* (*set* (*alphabetei A*)) (*set* (*initialei A*))
(*set* (*transitionei A*)) (*set* (*acceptingei A*))

lemma *nbaei-nbae-id-param*[*param*]: (*nbaei-nbae*, *id*) ∈ ⟨*L*, *S*⟩ *nbaei-nbae-rel* →
⟨*L*, *S*⟩ *nbae-rel*
⟨*proof*⟩

abbreviation *transitions L S s* ≡ $\bigcup$ *a* ∈ *L*. $\bigcup$ *p* ∈ *S*. {*p*} × {*a*} × *s a p*
abbreviation *succs T a p* ≡ (*T* `` {*p*}) `` {*a*}

definition *nba-nbae* **where** *nba-nbae A* ≡ *nbae* (*alphabet A*) (*initial A*)
(*transitions* (*alphabet A*) (*nodes A*) (*transition A*)) (*Set.filter* (*accepting A*)

(*nodes A*))
definition *nbae-nba* **where** *nbae-nba A* ≡ *nba* (*alphabete A*) (*initiale A*) (*succs* (*transitione A*)) (λ *p*. *p* ∈ *acceptinge A*)

lemma *nba-nbae-param*[*param*]: (*nba-nbae*, *nba-nbae*) ∈ ⟨*L*, *S*⟩ *nba-rel* → ⟨*L*, *S*⟩ *nbae-rel*
⟨*proof*⟩
lemma *nbae-nba-param*[*param*]:
assumes *bijective L bijective S*
shows (*nbae-nba*, *nbae-nba*) ∈ ⟨*L*, *S*⟩ *nbae-rel* → ⟨*L*, *S*⟩ *nba-rel*
⟨*proof*⟩

lemma *nbae-nba-nba-nbae-param*[*param*]:
((*nbae-nba* ∘ *nba-nbae*) *A*, *id A*) ∈ ⟨*Id-on* (*alphabet A*), *Id-on* (*nodes A*)⟩ *nba-rel*
⟨*proof*⟩

definition *nbaei-nba-rel* **where**
[*to-relAPP*]: *nbaei-nba-rel L S* ≡ {(*Ae*, *A*). (*nbae-nba* (*nbaei-nbae Ae*), *A*) ∈ ⟨*L*, *S*⟩ *nba-rel*}
lemma *nbaei-nba-id*[*param*]: (*nbae-nba* ∘ *nbaei-nbae*, *id*) ∈ ⟨*L*, *S*⟩ *nbaei-nba-rel* → ⟨*L*, *S*⟩ *nba-rel*
⟨*proof*⟩

schematic-goal *nbae-nba-impl*:
assumes [*autoref-rules*]: (*leq*, *HOL.eq*) ∈ *L* → *L* → *bool-rel*
assumes [*autoref-rules*]: (*seq*, *HOL.eq*) ∈ *S* → *S* → *bool-rel*
shows (*?f*, *nbae-nba*) ∈ ⟨*L*, *S*⟩ *nbaei-nbae-rel* → ⟨*L*, *S*⟩ *nbai-nba-rel*
⟨*proof*⟩
concrete-definition *nbae-nba-impl* **uses** *nbae-nba-impl*
lemma *nbae-nba-impl-refine*[*autoref-rules*]:
assumes *GEN-OP leq HOL.eq* (*L* → *L* → *bool-rel*)
assumes *GEN-OP seq HOL.eq* (*S* → *S* → *bool-rel*)
shows (*nbae-nba-impl leq seq*, *nbae-nba*) ∈ ⟨*L*, *S*⟩ *nbaei-nbae-rel* → ⟨*L*, *S*⟩ *nbai-nba-rel*
⟨*proof*⟩

end

43 Explore and Enumerate Nodes of Nondeterministic Büchi Automata

theory *NBA-Translate*
imports *NBA-Explicit*
begin

43.1 Syntax

no-syntax *-do-let* :: [*pttrn*, *'a*] ⇒ *do-bind* ((*2let* - =/ -) [*1000*, *13*] *13*)

syntax *-do-let* :: [*pttrn*, *'a*] $\Rightarrow$ *do-bind* ((*2let - =/ -*) *13*)

44 Image on Explicit Automata

definition *nbae-image* **where** *nbae-image f A* $\equiv$ *nbae* (*alphabete A*) (*f ' initiale A*)
((λ (*p*, *a*, *q*). (*f p*, *a*, *f q*)) *' transitione A*) (*f ' acceptinge A*)

lemma *nbae-image-param*[*param*]: (*nbae-image*, *nbae-image*) ∈ (*S* → *T*) → ⟨*L*, *S*⟩ *nbae-rel* → ⟨*L*, *T*⟩ *nbae-rel*
⟨*proof*⟩

lemma *nbae-image-id*[*simp*]: *nbae-image id* = *id* ⟨*proof*⟩
lemma *nbae-image-nba-nbae*: *nbae-image f* (*nba-nbae A*) = *nbae*
(*alphabet A*) (*f ' initial A*)
(⋃ *p* ∈ *nodes A*. ⋃ *a* ∈ *alphabet A*. *f '* {*p*} × {*a*} × *f ' transition A a p*)
(*f '* {*p* ∈ *nodes A*. *accepting A p*})
⟨*proof*⟩

45 Exploration and Translation

definition *trans-spec* **where**
trans-spec A f ≡ ⋃ *p* ∈ *nodes A*. ⋃ *a* ∈ *alphabet A*. *f '* {*p*} × {*a*} × *f ' transition A a p*

definition *trans-algo* **where**
trans-algo N L S f ≡
FOREACH N (λ *p T*. *do* {
ASSERT (*p* ∈ *N*);
FOREACH L (λ *a T*. *do* {
ASSERT (*a* ∈ *L*);
FOREACH (*S a p*) (λ *q T*. *do* {
ASSERT (*q* ∈ *S a p*);
ASSERT ((*f p*, *a*, *f q*) ∉ *T*);
RETURN (*insert* (*f p*, *a*, *f q*) *T*) }
) *T* }
) *T* }
) {}

lemma *trans-algo-refine*:
assumes *finite* (*nodes A*) *finite* (*alphabet A*) *inj-on f* (*nodes A*)
assumes *N* = *nodes A* *L* = *alphabet A* *S* = *transition A*
shows (*trans-algo N L S f*, *SPEC* (*HOL.eq* (*trans-spec A f*))) ∈ ⟨*Id*⟩ *nres-rel*
⟨*proof*⟩

definition *nba-image* :: ($'state_1 \Rightarrow 'state_2$) $\Rightarrow$ ($'label$, $'state_1$) *nba* $\Rightarrow$ ($'label$, $'state_2$) *nba* **where**

nba-image f A ≡ *nba*
(*alphabet A*)
(*f* ‘ *initial A*)
(λ *a p*. *f* ‘ *transition A a* (*inv-into* (*nodes A*) *f p*))
(λ *p*. *accepting A* (*inv-into* (*nodes A*) *f p*))

lemma *nba-image-rel*[*param*]:
assumes *inj-on f* (*nodes A*)
shows (*A*, *nba-image f A*) ∈ ⟨*Id-on* (*alphabet A*), *br f* (λ *p*. *p* ∈ *nodes A*)⟩ *nba-rel*
⟨*proof*⟩

lemma *nba-image-nodes*[*simp*]:
assumes *inj-on f* (*nodes A*)
shows *nodes* (*nba-image f A*) = *f* ‘ *nodes A*
⟨*proof*⟩
lemma *nba-image-language*[*simp*]:
assumes *inj-on f* (*nodes A*)
shows *language* (*nba-image f A*) = *language A*
⟨*proof*⟩

lemma *nba-image-nbae*:
assumes *inj-on f* (*nodes A*)
shows *nbae-image f* (*nba-nbae A*) = *nba-nbae* (*nba-image f A*)
⟨*proof*⟩

definition *op-translate* :: (*'label*, *'state*) *nba* ⇒ (*'label*, *nat*) *nbae nres* **where**
op-translate A ≡ *SPEC* (λ *B*. ∃ *f*. *inj-on f* (*nodes A*) ∧ *B* = *nba-nbae* (*nba-image f A*))

lemma *op-translate-language*:
assumes (*RETURN Ai*, *op-translate A*) ∈ ⟨⟨*Id*, *nat-rel*⟩ *nbaei-nbae-rel*⟩ *nres-rel*
shows *language* (*nbae-nba* (*nbaei-nbae Ai*)) = *language A*
⟨*proof*⟩

schematic-goal *to-nbaei-impl*:
fixes *S* :: (*'statei* × *'state*) *set*
assumes [*simp*]: *finite* (*nodes A*)
assumes [*autoref-ga-rules*]: *is-bounded-hashcode S seq bhc*
assumes [*autoref-ga-rules*]: *is-valid-def-hm-size TYPE*(*'statei*) *hms*
assumes [*autoref-rules*]: (*seq*, *HOL.eq*) ∈ *S* → *S* → *bool-rel*
assumes [*autoref-rules*]: (*Ai*, *A*) ∈ ⟨*L*, *S*⟩ *nbai-nba-rel*
shows (*?f* :: *?'a*, *do* {
let N = *nodes A*;
f ← *op-set-enumerate N*;

$ASSERT\ (dom\ f = N)$;
$ASSERT\ (\forall\ p \in initial\ A.\ f\ p \neq None)$;
$ASSERT\ (\forall\ a \in alphabet\ A.\ \forall\ p \in dom\ f.\ \forall\ q \in transition\ A\ a\ p.\ f\ q \neq None)$;
$T \leftarrow trans\text{-}algo\ N\ (alphabet\ A)\ (transition\ A)\ (\lambda\ x.\ the\ (f\ x))$;
$RETURN\ (nbae\ (alphabet\ A)\ ((\lambda\ x.\ the\ (f\ x))\ `\ initial\ A)\ T$
$((\lambda\ x.\ the\ (f\ x))\ `\ \{p \in N.\ accepting\ A\ p\}))$
$\})\in\ ?R$
$\langle proof \rangle$
concrete-definition *to-nbaei-impl* **uses** *to-nbaei-impl*

context
begin

interpretation *autoref-syn* $\langle proof \rangle$

lemma *to-nbaei-impl-refine*[*autoref-rules*]:
fixes S :: $('statei \times 'state)\ set$
assumes $SIDE\text{-}PRECOND\ (finite\ (nodes\ A))$
assumes $SIDE\text{-}GEN\text{-}ALGO\ (is\text{-}bounded\text{-}hashcode\ S\ seq\ bhc)$
assumes $SIDE\text{-}GEN\text{-}ALGO\ (is\text{-}valid\text{-}def\text{-}hm\text{-}size\ TYPE('statei)\ hms)$
assumes $GEN\text{-}OP\ seq\ HOL.eq\ (S \rightarrow S \rightarrow bool\text{-}rel)$
assumes $(Ai,\ A) \in \langle L,\ S\rangle\ nbai\text{-}nba\text{-}rel$
shows $(RETURN\ (to\text{-}nbaei\text{-}impl\ seq\ bhc\ hms\ Ai),$
$(OP\ op\text{-}translate\ :::\ \langle L,\ S\rangle\ nbai\text{-}nba\text{-}rel \rightarrow \langle\langle L,\ nat\text{-}rel\rangle\ nbaei\text{-}nbae\text{-}rel\rangle\ nres\text{-}rel)\ \$\ A) \in$
$\langle\langle L,\ nat\text{-}rel\rangle\ nbaei\text{-}nbae\text{-}rel\rangle\ nres\text{-}rel$
$\langle proof \rangle$

end

end

46 Connecting Nondeterministic Generalized Büchi Automata to CAVA Automata Structures

theory *NGBA-Graphs*
imports
NGBA
CAVA-Automata.Automata-Impl
begin

no-notation *build* (**infixr** ## *65*)

46.1 Regular Graphs

definition $ngba\text{-}g$ :: $('label,\ 'state)\ ngba \Rightarrow 'state\ graph\text{-}rec$ **where**
$ngba\text{-}g\ A \equiv (\!|\ g\text{-}V = UNIV,\ g\text{-}E = E\text{-}of\text{-}succ\ (successors\ A),\ g\text{-}V0 = initial\ A$

$\|)$

lemma *ngba-g-graph*[*simp*]: *graph* (*ngba-g A*) ⟨*proof*⟩

lemma *ngba-g-V0*: *g-V0* (*ngba-g A*) = *initial A* ⟨*proof*⟩
lemma *ngba-g-E-rtrancl*: $(g\text{-}E\ (ngba\text{-}g\ A))^* = \{(p, q).\ q \in reachable\ A\ p\}$
⟨*proof*⟩

lemma *ngba-g-rtrancl-path*: $(g\text{-}E\ (ngba\text{-}g\ A))^* = \{(p, target\ r\ p)\ |r\ p.\ NGBA.path\ A\ r\ p\}$
⟨*proof*⟩
lemma *ngba-g-trancl-path*: $(g\text{-}E\ (ngba\text{-}g\ A))^+ = \{(p, target\ r\ p)\ |r\ p.\ NGBA.path\ A\ r\ p \wedge r \neq []\}$
⟨*proof*⟩

lemma *ngba-g-ipath-run*:
assumes *ipath* (*g-E* (*ngba-g A*)) *r*
obtains *w*
where *run A* (*w* ||| *smap* (*r* ∘ *Suc*) *nats*) (*r 0*)
⟨*proof*⟩
lemma *ngba-g-run-ipath*:
assumes *run A* (*w* ||| *r*) *p*
shows *ipath* (*g-E* (*ngba-g A*)) (*snth* (*p* ## *r*))
⟨*proof*⟩

46.2 Indexed Generalized Büchi Graphs

definition *ngba-acc* :: *'state pred gen* ⇒ *'state* ⇒ *nat set* **where**
ngba-acc cs p ≡ $\{k \in \{0\ ..<\ length\ cs\}.\ (cs\ !\ k)\ p\}$

lemma *ngba-acc-param*[*param*]: (*ngba-acc*, *ngba-acc*) ∈ ⟨*S* → *bool-rel*⟩ *list-rel* → *S* → ⟨*nat-rel*⟩ *set-rel*
⟨*proof*⟩

definition *ngba-igbg* :: (*'label*, *'state*) *ngba* ⇒ *'state igb-graph-rec* **where**
ngba-igbg A ≡ *graph-rec.extend* (*ngba-g A*) (| *igbg-num-acc* = *length* (*accepting A*), *igbg-acc* = *ngba-acc* (*accepting A*) |)

lemma *acc-run-language*:
assumes *igb-graph* (*ngba-igbg A*)
shows *Ex* (*igb-graph.is-acc-run* (*ngba-igbg A*)) ⟷ *language A* ≠ {}
⟨*proof*⟩

end

47 Relations on Nondeterministic Generalized Büchi Automata

theory *NGBA-Refine*
imports
NGBA
../../Transition-Systems/Transition-System-Refine
begin

definition *ngba-rel* :: $('label_1 \times 'label_2)$ *set* $\Rightarrow$ $('state_1 \times 'state_2)$ *set* $\Rightarrow$
$(('label_1, 'state_1)$ *ngba* $\times$ $('label_2, 'state_2)$ *ngba*$)$ *set* **where**
[*to-relAPP*]: *ngba-rel* L S $\equiv$ $\{(A_1, A_2)$.
(*alphabet* A_1, *alphabet* A_2) $\in$ $\langle L\rangle$ *set-rel* $\wedge$
(*initial* A_1, *initial* A_2) $\in$ $\langle S\rangle$ *set-rel* $\wedge$
(*transition* A_1, *transition* A_2) $\in$ $L \rightarrow S \rightarrow \langle S\rangle$ *set-rel* $\wedge$
(*accepting* A_1, *accepting* A_2) $\in$ $\langle S \rightarrow$ *bool-rel*$\rangle$ *list-rel*$\}$

lemma *ngba-param*[*param*]:
(*ngba*, *ngba*) $\in$ $\langle L\rangle$ *set-rel* $\rightarrow$ $\langle S\rangle$ *set-rel* $\rightarrow$ $(L \rightarrow S \rightarrow \langle S\rangle$ *set-rel*$)$ $\rightarrow$ $\langle S \rightarrow$ *bool-rel*$\rangle$ *list-rel* $\rightarrow$
$\langle L, S\rangle$ *ngba-rel*
(*alphabet*, *alphabet*) $\in$ $\langle L, S\rangle$ *ngba-rel* $\rightarrow$ $\langle L\rangle$ *set-rel*
(*initial*, *initial*) $\in$ $\langle L, S\rangle$ *ngba-rel* $\rightarrow$ $\langle S\rangle$ *set-rel*
(*transition*, *transition*) $\in$ $\langle L, S\rangle$ *ngba-rel* $\rightarrow L \rightarrow S \rightarrow \langle S\rangle$ *set-rel*
(*accepting*, *accepting*) $\in$ $\langle L, S\rangle$ *ngba-rel* $\rightarrow$ $\langle S \rightarrow$ *bool-rel*$\rangle$ *list-rel*
$\langle$*proof*$\rangle$

lemma *ngba-rel-id*[*simp*]: $\langle Id, Id\rangle$ *ngba-rel* $= Id$ $\langle$*proof*$\rangle$

lemma *enableds-param*[*param*]: (*ngba.enableds*, *ngba.enableds*) $\in$ $\langle L, S\rangle$ *ngba-rel* $\rightarrow S \rightarrow \langle L \times_r S\rangle$ *set-rel*
$\langle$*proof*$\rangle$
lemma *paths-param*[*param*]: (*ngba.paths*, *ngba.paths*) $\in$ $\langle L, S\rangle$ *ngba-rel* $\rightarrow S \rightarrow$ $\langle\langle L \times_r S\rangle$ *list-rel*$\rangle$ *set-rel*
$\langle$*proof*$\rangle$
lemma *runs-param*[*param*]: (*ngba.runs*, *ngba.runs*) $\in$ $\langle L, S\rangle$ *ngba-rel* $\rightarrow S \rightarrow$ $\langle\langle L \times_r S\rangle$ *stream-rel*$\rangle$ *set-rel*
$\langle$*proof*$\rangle$

lemma *reachable-param*[*param*]: (*reachable*, *reachable*) $\in$ $\langle L, S\rangle$ *ngba-rel* $\rightarrow S \rightarrow$ $\langle S\rangle$ *set-rel*
$\langle$*proof*$\rangle$
lemma *nodes-param*[*param*]: (*nodes*, *nodes*) $\in$ $\langle L, S\rangle$ *ngba-rel* $\rightarrow$ $\langle S\rangle$ *set-rel*
$\langle$*proof*$\rangle$

lemma *gen-param*[*param*]: (*gen*, *gen*) $\in$ $(A \rightarrow B \rightarrow$ *bool-rel*$)$ $\rightarrow$ $\langle A\rangle$ *list-rel* $\rightarrow B$ $\rightarrow$ *bool-rel*
$\langle$*proof*$\rangle$

lemma *language-param*[*param*]: (*language*, *language*) $\in \langle L, S\rangle$ *ngba-rel* $\to \langle\langle L\rangle$ *stream-rel*$\rangle$ *set-rel*
$\langle$*proof*$\rangle$

end

48 Implementation of Nondeterministic Generalized Büchi Automata

theory *NGBA-Implement*
imports
NGBA-Refine
../../Basic/Implement
begin

consts *i-ngba-scheme* :: *interface* $\Rightarrow$ *interface* $\Rightarrow$ *interface*

context
begin

interpretation *autoref-syn* $\langle$*proof*$\rangle$

lemma *ngba-scheme-itype*[*autoref-itype*]:
ngba $::_i \langle L\rangle_i$ *i-set* $\to_i \langle S\rangle_i$ *i-set* $\to_i$ ($L \to_i S \to_i \langle S\rangle_i$ *i-set*) $\to_i \langle\langle S\rangle_i$ *i-set*$\rangle_i$ *i-list* $\to_i$
$\langle L, S\rangle_i$ *i-ngba-scheme*
alphabet $::_i \langle L, S\rangle_i$ *i-ngba-scheme* $\to_i \langle L\rangle_i$ *i-set*
initial $::_i \langle L, S\rangle_i$ *i-ngba-scheme* $\to_i \langle S\rangle_i$ *i-set*
transition $::_i \langle L, S\rangle_i$ *i-ngba-scheme* $\to_i L \to_i S \to_i \langle S\rangle_i$ *i-set*
accepting $::_i \langle L, S\rangle_i$ *i-ngba-scheme* $\to_i \langle\langle S\rangle_i$ *i-set*$\rangle_i$ *i-list*
$\langle$*proof*$\rangle$

end

datatype ('*label*, '*state*) *ngbai* = *ngbai*
(*alphabeti*: '*label list*)
(*initiali*: '*state list*)
(*transitioni*: '*label* $\Rightarrow$ '*state* $\Rightarrow$ '*state list*)
(*acceptingi*: ('*state* $\Rightarrow$ *bool*) *list*)

definition *ngbai-rel* :: ($'label_1 \times 'label_2$) *set* $\Rightarrow$ ($'state_1 \times 'state_2$) *set* $\Rightarrow$
(($'label_1$, $'state_1$) *ngbai* $\times$ ($'label_2$, $'state_2$) *ngbai*) *set* **where**
[*to-relAPP*]: *ngbai-rel* L $S \equiv \{(A_1, A_2)$.
(*alphabeti* A_1, *alphabeti* A_2) $\in \langle L\rangle$ *list-rel* $\wedge$
(*initiali* A_1, *initiali* A_2) $\in \langle S\rangle$ *list-rel* $\wedge$
(*transitioni* A_1, *transitioni* A_2) $\in L \to S \to \langle S\rangle$ *list-rel* $\wedge$
(*acceptingi* A_1, *acceptingi* A_2) $\in \langle S \to$ *bool-rel*$\rangle$ *list-rel*$\}$

lemma *ngbai-param*[*param*]:
$(ngbai,\ ngbai) \in \langle L\rangle\ list\text{-}rel \to \langle S\rangle\ list\text{-}rel \to (L \to S \to \langle S\rangle\ list\text{-}rel) \to \langle S \to bool\text{-}rel\rangle\ list\text{-}rel \to \langle L,\ S\rangle\ ngbai\text{-}rel$
$(alphabeti,\ alphabeti) \in \langle L,\ S\rangle\ ngbai\text{-}rel \to \langle L\rangle\ list\text{-}rel$
$(initiali,\ initiali) \in \langle L,\ S\rangle\ ngbai\text{-}rel \to \langle S\rangle\ list\text{-}rel$
$(transitioni,\ transitioni) \in \langle L,\ S\rangle\ ngbai\text{-}rel \to L \to S \to \langle S\rangle\ list\text{-}rel$
$(acceptingi,\ acceptingi) \in \langle L,\ S\rangle\ ngbai\text{-}rel \to \langle S \to bool\text{-}rel\rangle\ list\text{-}rel$
⟨*proof*⟩

definition *ngbai-ngba-rel* :: $('label_1 \times 'label_2)\ set \Rightarrow ('state_1 \times 'state_2)\ set \Rightarrow (('label_1,\ 'state_1)\ ngbai \times ('label_2,\ 'state_2)\ ngba)\ set$ **where**
[*to-relAPP*]: $ngbai\text{-}ngba\text{-}rel\ L\ S \equiv \{(A_1,\ A_2).$
$(alphabeti\ A_1,\ alphabet\ A_2) \in \langle L\rangle\ list\text{-}set\text{-}rel\ \wedge$
$(initiali\ A_1,\ initial\ A_2) \in \langle S\rangle\ list\text{-}set\text{-}rel\ \wedge$
$(transitioni\ A_1,\ transition\ A_2) \in L \to S \to \langle S\rangle\ list\text{-}set\text{-}rel\ \wedge$
$(acceptingi\ A_1,\ accepting\ A_2) \in \langle S \to bool\text{-}rel\rangle\ list\text{-}rel\}$

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of ngbai-ngba-rel i-ngba-scheme*]

lemma *ngbai-ngba-param*[*param*, *autoref-rules*]:
$(ngbai,\ ngba) \in \langle L\rangle\ list\text{-}set\text{-}rel \to \langle S\rangle\ list\text{-}set\text{-}rel \to (L \to S \to \langle S\rangle\ list\text{-}set\text{-}rel) \to \langle S \to bool\text{-}rel\rangle\ list\text{-}rel \to \langle L,\ S\rangle\ ngbai\text{-}ngba\text{-}rel$
$(alphabeti,\ alphabet) \in \langle L,\ S\rangle\ ngbai\text{-}ngba\text{-}rel \to \langle L\rangle\ list\text{-}set\text{-}rel$
$(initiali,\ initial) \in \langle L,\ S\rangle\ ngbai\text{-}ngba\text{-}rel \to \langle S\rangle\ list\text{-}set\text{-}rel$
$(transitioni,\ transition) \in \langle L,\ S\rangle\ ngbai\text{-}ngba\text{-}rel \to L \to S \to \langle S\rangle\ list\text{-}set\text{-}rel$
$(acceptingi,\ accepting) \in \langle L,\ S\rangle\ ngbai\text{-}ngba\text{-}rel \to \langle S \to bool\text{-}rel\rangle\ list\text{-}rel$
⟨*proof*⟩

definition *ngbai-ngba* :: $('label,\ 'state)\ ngbai \Rightarrow ('label,\ 'state)\ ngba$ **where**
$ngbai\text{-}ngba\ A \equiv ngba\ (set\ (alphabeti\ A))\ (set\ (initiali\ A))\ (\lambda\ a\ p.\ set\ (transitioni\ A\ a\ p))\ (acceptingi\ A)$
definition *ngbai-invar* :: $('label,\ 'state)\ ngbai \Rightarrow bool$ **where**
$ngbai\text{-}invar\ A \equiv distinct\ (alphabeti\ A) \wedge distinct\ (initiali\ A) \wedge (\forall\ a\ p.\ distinct\ (transitioni\ A\ a\ p))$

lemma *ngbai-ngba-id-param*[*param*]: $(ngbai\text{-}ngba,\ id) \in \langle L,\ S\rangle\ ngbai\text{-}ngba\text{-}rel \to \langle L,\ S\rangle\ ngba\text{-}rel$
⟨*proof*⟩

lemma *ngbai-ngba-br*: $\langle Id,\ Id\rangle\ ngbai\text{-}ngba\text{-}rel = br\ ngbai\text{-}ngba\ ngbai\text{-}invar$
⟨*proof*⟩

end
theory *Degeneralization-Refine*
imports *Degeneralization Refine*
begin

lemma *degen-param*[*param*]: (*degen*, *degen*) ∈ ⟨*S* → *bool-rel*⟩ *list-rel* → *S* $\times_r$ *nat-rel* → *bool-rel*
⟨*proof*⟩

lemma *count-param*[*param*]: (*Degeneralization.count*, *Degeneralization.count*) ∈
⟨*A* → *bool-rel*⟩ *list-rel* → *A* → *nat-rel* → *nat-rel*
⟨*proof*⟩

end

49 Algorithms on Nondeterministic Generalized Büchi Automata

theory *NGBA-Algorithms*
imports
NGBA-Graphs
NGBA-Implement
NBA-Combine
NBA-Algorithms
Degeneralization-Refine
begin

49.1 Operations

definition *op-language-empty* **where** [*simp*]: *op-language-empty A* ≡ *NGBA.language A* = {}

lemmas [*autoref-op-pat*] = *op-language-empty-def*[*symmetric*]

49.2 Implementations

context
begin

interpretation *autoref-syn* ⟨*proof*⟩

lemma *ngba-g-ahs*: *ngba-g A* = (| *g-V* = *UNIV*, *g-E* = *E-of-succ* (λ *p*. *CAST*
((⋃ *a* ∈ *ngba.alphabet A*. *ngba.transition A a p* ::: ⟨*S*⟩ *list-set-rel*) ::: ⟨*S*⟩ *ahs-rel bhc*)),
g-V0 = *ngba.initial A* |)
⟨*proof*⟩

schematic-goal *ngbai-gi*:
notes [*autoref-ga-rules*] = *map2set-to-list*
fixes *S* :: (′*statei* × ′*state*) *set*
assumes [*autoref-ga-rules*]: *is-bounded-hashcode S seq bhc*
assumes [*autoref-ga-rules*]: *is-valid-def-hm-size TYPE*(′*statei*) *hms*
assumes [*autoref-rules*]: (*seq*, *HOL.eq*) ∈ *S* → *S* → *bool-rel*
assumes [*autoref-rules*]: (*Ai*, *A*) ∈ ⟨*L*, *S*⟩ *ngbai-ngba-rel*

shows (*?f* :: *?'a*, *RETURN* (*ngba-g A*)) ∈ *?A*
⟨*proof*⟩
concrete-definition *ngbai-gi* **uses** *ngbai-gi*
lemma *ngbai-gi-refine*[*autoref-rules*]:
fixes *S* :: (*'statei* × *'state*) *set*
assumes *SIDE-GEN-ALGO* (*is-bounded-hashcode S seq bhc*)
assumes *SIDE-GEN-ALGO* (*is-valid-def-hm-size TYPE*(*'statei*) *hms*)
assumes *GEN-OP seq HOL.eq* (*S* → *S* → *bool-rel*)
shows (*NGBA-Algorithms.ngbai-gi seq bhc hms*, *ngba-g*) ∈
⟨*L*, *S*⟩ *ngbai-ngba-rel* → ⟨*unit-rel*, *S*⟩ *g-impl-rel-ext*
⟨*proof*⟩

schematic-goal *ngba-nodes*:
fixes *S* :: (*'statei* × *'state*) *set*
assumes [*simp*]: *finite* ((*g-E* (*ngba-g A*))* ‘‘ *g-V0* (*ngba-g A*))
assumes [*autoref-ga-rules*]: *is-bounded-hashcode S seq bhc*
assumes [*autoref-ga-rules*]: *is-valid-def-hm-size TYPE*(*'statei*) *hms*
assumes [*autoref-rules*]: (*seq*, *HOL.eq*) ∈ *S* → *S* → *bool-rel*
assumes [*autoref-rules*]: (*Ai*, *A*) ∈ ⟨*L*, *S*⟩ *ngbai-ngba-rel*
shows (*?f* :: *?'a*, *op-reachable* (*ngba-g A*)) ∈ *?R* ⟨*proof*⟩
concrete-definition *ngba-nodes* **uses** *ngba-nodes*
lemma *ngba-nodes-refine*[*autoref-rules*]:
fixes *S* :: (*'statei* × *'state*) *set*
assumes *SIDE-PRECOND* (*finite* (*NGBA.nodes A*))
assumes *SIDE-GEN-ALGO* (*is-bounded-hashcode S seq bhc*)
assumes *SIDE-GEN-ALGO* (*is-valid-def-hm-size TYPE*(*'statei*) *hms*)
assumes *GEN-OP seq HOL.eq* (*S* → *S* → *bool-rel*)
assumes (*Ai*, *A*) ∈ ⟨*L*, *S*⟩ *ngbai-ngba-rel*
shows (*NGBA-Algorithms.ngba-nodes seq bhc hms Ai*,
(*OP NGBA.nodes* ::: ⟨*L*, *S*⟩ *ngbai-ngba-rel* → ⟨*S*⟩ *ahs-rel bhc*) $ *A*) ∈ ⟨*S*⟩ *ahs-rel bhc*
⟨*proof*⟩

lemma *ngba-igbg-ahs*: *ngba-igbg A* = (| *g-V* = *UNIV*, *g-E* = *E-of-succ* (λ *p*. *CAST*
((⋃ *a* ∈ *NGBA.alphabet A*. *NGBA.transition A a p* ::: ⟨*S*⟩ *list-set-rel*) ::: ⟨*S*⟩ *ahs-rel bhc*)), *g-V0* = *NGBA.initial A*,
igbg-num-acc = *length* (*NGBA.accepting A*), *igbg-acc* = *ngba-acc* (*NGBA.accepting A*) |)
⟨*proof*⟩

definition *ngba-acc-bs cs p* ≡ *fold* (λ (*k*, *c*) *bs*. *if c p then bs-insert k bs else bs*) (*List.enumerate 0 cs*) (*bs-empty* ())

lemma *ngba-acc-bs-empty*[*simp*]: *ngba-acc-bs* [] *p* = *bs-empty* () ⟨*proof*⟩
lemma *ngba-acc-bs-insert*[*simp*]:
assumes *c p*
shows *ngba-acc-bs* (*cs* @ [*c*]) *p* = *bs-insert* (*length cs*) (*ngba-acc-bs cs p*)
⟨*proof*⟩

lemma *ngba-acc-bs-skip*[*simp*]:
  assumes ¬ *c p*
  shows *ngba-acc-bs* (*cs* @ [*c*]) *p* = *ngba-acc-bs cs p*
  ⟨*proof*⟩

lemma *ngba-acc-bs-correct*[*simp*]: *bs-α* (*ngba-acc-bs cs p*) = *ngba-acc cs p*
⟨*proof*⟩

lemma *ngba-acc-impl-bs*[*autoref-rules*]: (*ngba-acc-bs*, *ngba-acc*) ∈ ⟨*S* → *bool-rel*⟩ *list-rel* → *S* → ⟨*nat-rel*⟩ *bs-set-rel*
⟨*proof*⟩

schematic-goal *ngbai-igbgi*:
  notes [*autoref-ga-rules*] = *map2set-to-list*
  fixes *S* :: (′*statei* × ′*state*) *set*
  assumes [*autoref-ga-rules*]: *is-bounded-hashcode S seq bhc*
  assumes [*autoref-ga-rules*]: *is-valid-def-hm-size TYPE*(′*statei*) *hms*
  assumes [*autoref-rules*]: (*seq*, *HOL.eq*) ∈ *S* → *S* → *bool-rel*
  assumes [*autoref-rules*]: (*Ai*, *A*) ∈ ⟨*L*, *S*⟩ *ngbai-ngba-rel*
  shows (*?f* :: *?′a*, *RETURN* (*ngba-igbg A*)) ∈ *?A*
  ⟨*proof*⟩
concrete-definition *ngbai-igbgi* **uses** *ngbai-igbgi*
lemma *ngbai-igbgi-refine*[*autoref-rules*]:
  fixes *S* :: (′*statei* × ′*state*) *set*
  assumes *SIDE-GEN-ALGO* (*is-bounded-hashcode S seq bhc*)
  assumes *SIDE-GEN-ALGO* (*is-valid-def-hm-size TYPE*(′*statei*) *hms*)
  assumes *GEN-OP seq HOL.eq* (*S* → *S* → *bool-rel*)
  shows (*NGBA-Algorithms.ngbai-igbgi seq bhc hms*, *ngba-igbg*) ∈
    ⟨*L*, *S*⟩ *ngbai-ngba-rel* → *igbg-impl-rel-ext unit-rel S*
  ⟨*proof*⟩

schematic-goal *ngba-language-empty*:
  fixes *S* :: (′*statei* × ′*state*) *set*
  assumes [*simp*]: *igb-fr-graph* (*ngba-igbg A*)
  assumes [*autoref-ga-rules*]: *is-bounded-hashcode S seq bhs*
  assumes [*autoref-ga-rules*]: *is-valid-def-hm-size TYPE*(′*statei*) *hms*
  assumes [*autoref-rules*]: (*seq*, *HOL.eq*) ∈ *S* → *S* → *bool-rel*
  assumes [*autoref-rules*]: (*Ai*, *A*) ∈ ⟨*L*, *S*⟩ *ngbai-ngba-rel*
  shows (*?f* :: *?′a*, *do* { *r* ← *op-find-lasso-spec* (*ngba-igbg A*); *RETURN* (*r* = *None*)}) ∈ *?A*
  ⟨*proof*⟩
concrete-definition *ngba-language-empty* **uses** *ngba-language-empty*
lemma *nba-language-empty-refine*[*autoref-rules*]:
  fixes *S* :: (′*statei* × ′*state*) *set*
  assumes *SIDE-PRECOND* (*finite* (*NGBA.nodes A*))
  assumes *SIDE-GEN-ALGO* (*is-bounded-hashcode S seq bhc*)
  assumes *SIDE-GEN-ALGO* (*is-valid-def-hm-size TYPE*(′*statei*) *hms*)
  assumes *GEN-OP seq HOL.eq* (*S* → *S* → *bool-rel*)
  assumes (*Ai*, *A*) ∈ ⟨*L*, *S*⟩ *ngbai-ngba-rel*

shows (*NGBA-Algorithms.ngba-language-empty seq bhc hms Ai*,
(*OP op-language-empty* ::: $\langle L, S\rangle$ *ngbai-ngba-rel* $\to$ *bool-rel*) \$ *A*) $\in$ *bool-rel*
⟨*proof*⟩

lemma *degeneralize-alt-def*: *degeneralize A* = *nba*
(*ngba.alphabet A*)
((λ *p*. (*p*, *0*)) ' *ngba.initial A*)
(λ *a* (*p*, *k*). (λ *q*. (*q*, *Degeneralization.count* (*ngba.accepting A*) *p k*)) ' *ngba.transition A a p*)
(*degen* (*ngba.accepting A*))
⟨*proof*⟩

schematic-goal *ngba-degeneralize*: (*?f* :: *?'a*, *degeneralize*) $\in$ *?R*
⟨*proof*⟩
concrete-definition *ngba-degeneralize* **uses** *ngba-degeneralize*
lemmas *ngba-degeneralize-refine*[*autoref-rules*] = *ngba-degeneralize.refine*

schematic-goal *nba-intersect'*:
assumes [*autoref-rules*]: (*seq*, *HOL.eq*) $\in L \to L \to$ *bool-rel*
shows (*?f*, *intersect'*) $\in$ $\langle L, S\rangle$ *nbai-nba-rel* $\to$ $\langle L, T\rangle$ *nbai-nba-rel* $\to$ $\langle L, S \times_r T\rangle$ *ngbai-ngba-rel*
⟨*proof*⟩
concrete-definition *nba-intersect'* **uses** *nba-intersect'*
lemma *nba-intersect'-refine*[*autoref-rules*]:
assumes *GEN-OP seq HOL.eq* ($L \to L \to$ *bool-rel*)
shows (*nba-intersect' seq*, *intersect'*) $\in$
$\langle L, S\rangle$ *nbai-nba-rel* $\to$ $\langle L, T\rangle$ *nbai-nba-rel* $\to$ $\langle L, S \times_r T\rangle$ *ngbai-ngba-rel*
⟨*proof*⟩

end

end

50 Nondeterministic Büchi Transition Automata

theory *NBTA*
imports ../*Nondeterministic*
begin

datatype (*'label*, *'state*) *nbta* = *nbta*
(*alphabet*: *'label set*)
(*initial*: *'state set*)
(*transition*: *'label* ⇒ *'state* ⇒ *'state set*)
(*accepting*: (*'state* × *'label* × *'state*) *pred*)

global-interpretation *nbta*: *automaton nbta alphabet initial transition accepting*
defines *path* = *nbta.path* **and** *run* = *nbta.run* **and** *reachable* = *nbta.reachable* **and** *nodes* = *nbta.nodes*
⟨*proof*⟩

global-interpretation *nbta*: *automaton-run nbta alphabet initial transition accepting*
λ *P w r p. infs P* (*p* ## *r* ||| *w* ||| *r*)
defines *language* = *nbta.language*
⟨*proof*⟩

abbreviation *target* **where** *target* ≡ *nbta.target*
abbreviation *states* **where** *states* ≡ *nbta.states*
abbreviation *trace* **where** *trace* ≡ *nbta.trace*
abbreviation *successors* **where** *successors* ≡ *nbta.successors* *TYPE*('*label*)

end

51 Nondeterministic Generalized Büchi Transition Automata

theory *NGBTA*
imports ../*Nondeterministic*
begin

datatype ('*label*, '*state*) *ngbta* = *ngbta*
(*alphabet*: '*label set*)
(*initial*: '*state set*)
(*transition*: '*label* ⇒ '*state* ⇒ '*state set*)
(*accepting*: ('*state* × '*label* × '*state*) *pred gen*)

global-interpretation *ngbta*: *automaton ngbta alphabet initial transition accepting*
defines *path* = *ngbta.path* **and** *run* = *ngbta.run* **and** *reachable* = *ngbta.reachable* **and** *nodes* = *ngbta.nodes*
⟨*proof*⟩
global-interpretation *ngbta*: *automaton-run ngbta alphabet initial transition accepting*
λ *P w r p. gen infs P* (*p* ## *r* ||| *w* ||| *r*)
defines *language* = *ngbta.language*
⟨*proof*⟩

abbreviation *target* **where** *target* ≡ *ngbta.target*
abbreviation *states* **where** *states* ≡ *ngbta.states*
abbreviation *trace* **where** *trace* ≡ *ngbta.trace*
abbreviation *successors* **where** *successors* ≡ *ngbta.successors* *TYPE*('*label*)

end

52 Nondeterministic Büchi Transition Automata Combinations

theory *NBTA-Combine*
imports *NBTA NGBTA*
begin

global-interpretation *degeneralization*: *automaton-degeneralization-run*
ngbta ngbta.alphabet ngbta.initial ngbta.transition ngbta.accepting λ *P w r p. gen infs P* (*p ## r* ||| *w* ||| *r*)
nbta nbta.alphabet nbta.initial nbta.transition nbta.accepting λ *P w r p. infs P* (*p ## r* ||| *w* ||| *r*)
id λ ((*p*, *k*), *a*, (*q*, *l*)). ((*p*, *a*, *q*), *k*)
defines *degeneralize* = *degeneralization.degeneralize*
⟨*proof*⟩

lemmas *degeneralize-language*[*simp*] = *degeneralization.degeneralize-language*[*folded NBTA.language-def*]
lemmas *degeneralize-nodes-finite*[*iff*] = *degeneralization.degeneralize-nodes-finite*[*folded NBTA.nodes-def*]

global-interpretation *intersection*: *automaton-intersection-run*
nbta nbta.alphabet nbta.initial nbta.transition nbta.accepting λ *P w r p. infs P* (*p ## r* ||| *w* ||| *r*)
nbta nbta.alphabet nbta.initial nbta.transition nbta.accepting λ *P w r p. infs P* (*p ## r* ||| *w* ||| *r*)
ngbta ngbta.alphabet ngbta.initial ngbta.transition ngbta.accepting λ *P w r p. gen infs P* (*p ## r* ||| *w* ||| *r*)
λ c_1 c_2. [c_1 $\circ$ (λ ((p_1, p_2), *a*, (q_1, q_2)). (p_1, *a*, q_1)), c_2 $\circ$ (λ ((p_1, p_2), *a*, (q_1, q_2)). (p_2, *a*, q_2))]
defines *intersect′* = *intersection.product*
⟨*proof*⟩

lemmas *intersect′-language*[*simp*] = *intersection.product-language*[*folded NGBTA.language-def*]
lemmas *intersect′-nodes-finite*[*intro*] = *intersection.product-nodes-finite*[*folded NGBTA.nodes-def*]

global-interpretation *union*: *automaton-union-run*
nbta nbta.alphabet nbta.initial nbta.transition nbta.accepting λ *P w r p. infs P* (*p ## r* ||| *w* ||| *r*)
nbta nbta.alphabet nbta.initial nbta.transition nbta.accepting λ *P w r p. infs P* (*p ## r* ||| *w* ||| *r*)
nbta nbta.alphabet nbta.initial nbta.transition nbta.accepting λ *P w r p. infs P* (*p ## r* ||| *w* ||| *r*)
λ c_1 c_2 *m. case m of* (*Inl p*, *a*, *Inl q*) $\Rightarrow$ c_1 (*p*, *a*, *q*) | (*Inr p*, *a*, *Inr q*) $\Rightarrow$ c_2 (*p*, *a*, *q*)
defines *union* = *union.sum*
⟨*proof*⟩

lemmas *union-language* = *union.sum-language*
lemmas *union-nodes-finite* = *union.sum-nodes-finite*

abbreviation *intersect* **where** *intersect A B* ≡ *degeneralize* (*intersect′ A B*)

lemma *intersect-language*[*simp*]: *NBTA.language* (*intersect A B*) = *NBTA.language A* ∩ *NBTA.language B*
⟨*proof*⟩
lemma *intersect-nodes-finite*[*intro*]:
assumes *finite* (*NBTA.nodes A*) *finite* (*NBTA.nodes B*)
shows *finite* (*NBTA.nodes* (*intersect A B*))
⟨*proof*⟩

end