Verified Complete Test Strategies for Finite State Machines

Robert Sachtleben

March 17, 2025

Abstract

This entry provides executable formalisations of the following testing strategies based on finite state machines (FSM):

1. Strategies for language-equivalence testing on possibly nondeterministic and partial FSMs:
   - W-Method [1]
   - Wp-Method (based on a generalisation of [4] presented in [5])
   - HSI-Method [3]
   - H-Method [2]
   - SPY-Method [10]
   - SPYH-Method [11]
2. Strategies for reduction testing on possibly nondeterministic FSMs:
   - Adaptive state counting (as described in [6])

These strategies are implemented using generic frameworks which allow combining parts of strategies such as reaching and distinguishing of states or distributing traces over classes of convergent traces. Further details are given in the corresponding PhD thesis [8] and tools employing the code generated from this entry are available at https://bitbucket.org/RobertSachtleben/an-approach-for-the-verification-and-synthesis-of-complete.

In addition to formalising different algorithms, this entry differs from my previous entry [7] (see [9] for the corresponding paper) in using a revised representation of finite state machines and by a focus on executable definitions.

Contents

1 Utility Definitions and Properties

This file contains various definitions and lemmata not closely related to finite state machines or testing.

theory *Util*
imports *Main HOL−Library.FSet HOL−Library.Sublist HOL−Library.Mapping*
begin

1.1 Converting Sets to Maps

This subsection introduces a function *set-as-map* that transforms a set of $('a \times 'b)$ tuples to a map mapping each first value x of the contained tuples to all second values y such that (x,y) is contained in the set.

definition *set-as-map* :: $('a \times 'c)$ *set* $\Rightarrow$ $('a \Rightarrow 'c$ *set option*$)$ **where**
set-as-map s = $(\lambda\ x\ .\ if\ (\exists\ z\ .\ (x,z) \in s)\ then\ Some\ \{z\ .\ (x,z) \in s\}\ else\ None)$

lemma *set-as-map-code*[*code*] :
```
set-as-map (set xs) = (foldl (λ m (x,z) . case m x of
                                        None ⇒ m (x ↦ {z}) |
                                        Some zs ⇒ m (x ↦  (insert z zs)))
                              Map.empty
                              xs)
```
⟨*proof*⟩

abbreviation *member-option x ms* ≡ (*case ms of None* ⇒ *False* | *Some xs* ⇒ $x \in xs$)
notation *member-option* (‹(-$\in_o$-)› [*1000*] *1000*)

abbreviation(*input*) *lookup-with-default f d* ≡ (λ *x* . *case f x of None* ⇒ *d* | *Some xs* ⇒ *xs*)
abbreviation(*input*) *m2f f* ≡ *lookup-with-default f* {}

abbreviation(*input*) *lookup-with-default-by f g d* ≡ (λ *x* . *case f x of None* ⇒ *g d* | *Some xs* ⇒ *g xs*)
abbreviation(*input*) *m2f-by g f* ≡ *lookup-with-default-by f g* {}

lemma *m2f-by-from-m2f* :
(*m2f-by g f xs*) = *g* (*m2f f xs*)
⟨*proof*⟩

lemma *set-as-map-containment* :
assumes $(x,y) \in zs$
shows $y \in$ (*m2f* (*set-as-map zs*)) *x*
⟨*proof*⟩

lemma *set-as-map-elem* :
assumes $y \in$ *m2f* (*set-as-map xs*) *x*
shows $(x,y) \in xs$
⟨*proof*⟩

1.2 Utility Lemmata for existing functions on lists

1.2.1 Utility Lemmata for *find*

lemma *find-result-props* :
assumes *find* P *xs* = *Some* x
shows $x \in$ *set xs* **and** $P\ x$
⟨*proof*⟩

lemma *find-set* :
assumes *find* P *xs* = *Some* x
shows $x \in$ *set xs*
⟨*proof*⟩

lemma *find-condition* :
assumes *find* P *xs* = *Some* x
shows $P\ x$
⟨*proof*⟩

lemma *find-from* :
assumes $\exists\ x \in$ *set xs* . $P\ x$
shows *find* P *xs* $\neq$ *None*
⟨*proof*⟩

lemma *find-sort-containment* :
assumes *find* P (*sort xs*) = *Some* x
shows $x \in$ *set xs*
⟨*proof*⟩

lemma *find-sort-index* :
assumes *find* P *xs* = *Some* x
shows $\exists\ i <$ *length xs* . *xs* ! $i = x \wedge (\forall\ j < i\ .\ \neg\ P\ (xs\ !\ j))$
⟨*proof*⟩

lemma *find-sort-least* :
assumes *find* P (*sort xs*) = *Some* x
shows $\forall\ x' \in$ *set xs* . $x \leq x' \vee \neg\ P\ x'$
and $x = (LEAST\ x' \in set\ xs\ .\ P\ x')$
⟨*proof*⟩

1.2.2 Utility Lemmata for *filter*

lemma *filter-take-length* :
length (*filter* P (*take* i *xs*)) $\leq$ *length* (*filter* P *xs*)
⟨*proof*⟩

lemma *filter-double* :
assumes $x \in set\ (filter\ P1\ xs)$
and $P2\ x$
shows $x \in set\ (filter\ P2\ (filter\ P1\ xs))$
⟨*proof*⟩

lemma *filter-list-set* :
assumes $x \in set\ xs$
and $P\ x$
shows $x \in set\ (filter\ P\ xs)$
⟨*proof*⟩

lemma *filter-list-set-not-contained* :
assumes $x \in set\ xs$
and $\neg\ P\ x$
shows $x \notin set\ (filter\ P\ xs)$
⟨*proof*⟩

lemma *filter-map-elem* : $t \in set\ (map\ g\ (filter\ f\ xs)) \Longrightarrow \exists\ x \in set\ xs\ .\ f\ x \wedge t = g\ x$
⟨*proof*⟩

1.2.3 Utility Lemmata for *concat*

lemma *concat-map-elem* :
assumes $y \in set\ (concat\ (map\ f\ xs))$
obtains x **where** $x \in set\ xs$
and $y \in set\ (f\ x)$
⟨*proof*⟩

lemma *set-concat-map-sublist* :
assumes $x \in set\ (concat\ (map\ f\ xs))$
and $set\ xs \subseteq set\ xs'$
shows $x \in set\ (concat\ (map\ f\ xs'))$
⟨*proof*⟩

lemma *set-concat-map-elem* :
assumes $x \in set\ (concat\ (map\ f\ xs))$
shows $\exists\ x' \in set\ xs\ .\ x \in set\ (f\ x')$
⟨*proof*⟩

lemma *concat-replicate-length* : $length\ (concat\ (replicate\ n\ xs)) = n * (length\ xs)$
⟨*proof*⟩

1.3 Enumerating Lists

fun *lists-of-length* :: $'a\ list \Rightarrow nat \Rightarrow 'a\ list\ list$ **where**
$lists\text{-}of\text{-}length\ T\ 0 = [[]]\ |$
$lists\text{-}of\text{-}length\ T\ (Suc\ n) = concat\ (map\ (\lambda\ xs\ .\ map\ (\lambda\ x\ .\ x\#xs)\ T\ )\ (lists\text{-}of\text{-}length\ T\ n))$

lemma *lists-of-length-containment* :
assumes *set xs* ⊆ *set T*
and *length xs* = *n*
shows *xs* ∈ *set* (*lists-of-length T n*)
⟨*proof*⟩

lemma *lists-of-length-length* :
assumes *xs* ∈ *set* (*lists-of-length T n*)
shows *length xs* = *n*
⟨*proof*⟩

lemma *lists-of-length-elems* :
assumes *xs* ∈ *set* (*lists-of-length T n*)
shows *set xs* ⊆ *set T*
⟨*proof*⟩

lemma *lists-of-length-list-set* :
set (*lists-of-length xs k*) = {*xs'* . *length xs'* = *k* ∧ *set xs'* ⊆ *set xs*}
⟨*proof*⟩

1.3.1 Enumerating List Subsets

fun *generate-selector-lists* :: *nat* ⇒ *bool list list* **where**
generate-selector-lists k = *lists-of-length* [*False*,*True*] *k*

lemma *generate-selector-lists-set* :
set (*generate-selector-lists k*) = {(*bs* :: *bool list*) . *length bs* = *k*}
⟨*proof*⟩

lemma *selector-list-index-set*:
assumes *length ms* = *length bs*
shows *set* (*map fst* (*filter snd* (*zip ms bs*))) = { *ms* ! *i* | *i* . *i* < *length bs* ∧ *bs* ! *i*}
⟨*proof*⟩

lemma *selector-list-ex* :
assumes *set xs* ⊆ *set ms*
shows ∃ *bs* . *length bs* = *length ms* ∧ *set xs* = *set* (*map fst* (*filter snd* (*zip ms bs*)))
⟨*proof*⟩

1.3.2 Enumerating Choices from Lists of Lists

fun *generate-choices* :: ('*a* × ('*b list*)) *list* ⇒ ('*a* × '*b option*) *list list* **where**
generate-choices [] = [[]] |
generate-choices (*xys*#*xyss*) =
concat (*map* (λ *xy'* . *map* (λ *xys'* . *xy'* # *xys'*) (*generate-choices xyss*))

$((fst\ xys,\ None)\ \#\ (map\ (\lambda\ y\ .\ (fst\ xys,\ Some\ y))\ (snd\ xys))))$

lemma *concat-map-hd-tl-elem*:
assumes $hd\ cs \in set\ P1$
and $tl\ cs \in set\ P2$
and $length\ cs > 0$
shows $cs \in set\ (concat\ (map\ (\lambda\ xy'\ .\ map\ (\lambda\ xys'\ .\ xy'\ \#\ xys')\ P2)\ P1))$
⟨*proof*⟩

lemma *generate-choices-hd-tl* :
$cs \in set\ (generate\text{-}choices\ (xys\#xyss))$
$= (length\ cs = length\ (xys\#xyss)$
$\wedge\ fst\ (hd\ cs) = fst\ xys$
$\wedge\ ((snd\ (hd\ cs) = None \vee (snd\ (hd\ cs) \neq None \wedge the\ (snd\ (hd\ cs)) \in set\ (snd\ xys))))$
$\wedge\ (tl\ cs \in set\ (generate\text{-}choices\ xyss)))$
⟨*proof*⟩

lemma *list-append-idx-prop* :
$(\forall\ i\ .\ (i < length\ xs \longrightarrow P\ (xs\ !\ i)))$
$= (\forall\ j\ .\ ((j < length\ (ys@xs) \wedge j \geq length\ ys) \longrightarrow P\ ((ys@xs)\ !\ j)))$
⟨*proof*⟩

lemma *list-append-idx-prop2* :
assumes $length\ xs' = length\ xs$
and $length\ ys' = length\ ys$
shows $(\forall\ i\ .\ (i < length\ xs \longrightarrow P\ (xs\ !\ i)\ (xs'\ !\ i)))$
$= (\forall\ j\ .\ ((j < length\ (ys@xs) \wedge j \geq length\ ys) \longrightarrow P\ ((ys@xs)\ !\ j)\ ((ys'@xs')\ !\ j)))$
⟨*proof*⟩

lemma *generate-choices-idx* :
$cs \in set\ (generate\text{-}choices\ xyss)$
$= (length\ cs = length\ xyss$
$\wedge\ (\forall\ i < length\ cs\ .\ (fst\ (cs\ !\ i)) = (fst\ (xyss\ !\ i))$
$\wedge\ ((snd\ (cs\ !\ i)) = None$
$\vee\ ((snd\ (cs\ !\ i)) \neq None \wedge the\ (snd\ (cs\ !\ i)) \in set\ (snd\ (xyss\ !\ i))))))$
⟨*proof*⟩

1.4 Finding the Index of the First Element of a List Satisfying a Property

fun *find-index* :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow nat\ option$ **where**
$find\text{-}index\ f\ [] = None\ |$
$find\text{-}index\ f\ (x\#xs) = (if\ f\ x$
$then\ Some\ 0$
$else\ (case\ find\text{-}index\ f\ xs\ of\ Some\ k \Rightarrow Some\ (Suc\ k)\ |\ None \Rightarrow None))$

lemma *find-index-index* :
  assumes *find-index f xs* = *Some k*
  shows k < *length xs* **and** f (*xs* ! k) **and** $\bigwedge j . j < k \Longrightarrow \neg f$ (*xs* ! j)
⟨*proof*⟩

lemma *find-index-exhaustive* :
  assumes $\exists\ x \in$ *set xs* . $f\ x$
  shows *find-index f xs* $\neq$ *None*
  ⟨*proof*⟩

1.5 List Distinctness from Sorting

lemma *non-distinct-repetition-indices* :
  assumes $\neg$ *distinct xs*
  shows $\exists\ i\ j\ .\ i < j \land j <$ *length xs* $\land$ *xs* ! i = *xs* ! j
  ⟨*proof*⟩

lemma *non-distinct-repetition-indices-rev* :
  assumes $i < j$ **and** $j <$ *length xs* **and** *xs* ! i = *xs* ! j
  shows $\neg$ *distinct xs*
  ⟨*proof*⟩

lemma *ordered-list-distinct* :
  fixes *xs* :: (*'a*::*preorder*) *list*
  assumes $\bigwedge i$. *Suc i* < *length xs* $\Longrightarrow$ (*xs* ! i) < (*xs* ! (*Suc i*))
  shows *distinct xs*
⟨*proof*⟩

lemma *ordered-list-distinct-rev* :
  fixes *xs* :: (*'a*::*preorder*) *list*
  assumes $\bigwedge i$. *Suc i* < *length xs* $\Longrightarrow$ (*xs* ! i) > (*xs* ! (*Suc i*))
  shows *distinct xs*
⟨*proof*⟩

1.6 Calculating Prefixes and Suffixes

fun *suffixes* :: *'a list* ⇒ *'a list list* **where**
  suffixes [] = [[]] |
  suffixes (*x#xs*) = (*suffixes xs*) @ [*x#xs*]

lemma *suffixes-set* :
  set (*suffixes xs*) = {*zs* . ∃ *ys* . *ys*@*zs* = *xs*}
⟨*proof*⟩

lemma *prefixes-set* : *set* (*prefixes xs*) = {*xs′* . ∃ *xs″* . *xs′*@*xs″* = *xs*}
⟨*proof*⟩

fun *is-prefix* :: ′*a list* ⇒ ′*a list* ⇒ *bool* **where**
is-prefix [] - = *True* |
is-prefix (*x*#*xs*) [] = *False* |
is-prefix (*x*#*xs*) (*y*#*ys*) = (*x* = *y* ∧ *is-prefix xs ys*)

lemma *is-prefix-prefix* : *is-prefix xs ys* = (∃ *xs′* . *ys* = *xs*@*xs′*)
⟨*proof*⟩

fun *add-prefixes* :: ′*a list list* ⇒ ′*a list list* **where**
add-prefixes xs = *concat* (*map prefixes xs*)

lemma *add-prefixes-set* : *set* (*add-prefixes xs*) = {*xs′* . ∃ *xs″* . *xs′*@*xs″* ∈ *set xs*}
⟨*proof*⟩

lemma *prefixes-set-ob* :
assumes *xs* ∈ *set* (*prefixes xss*)
obtains *xs′* **where** *xss* = *xs*@*xs′*
⟨*proof*⟩

lemma *prefixes-finite* : *finite* { *x* ∈ *set* (*prefixes xs*) . *P x*}
⟨*proof*⟩

lemma *prefixes-set-Cons-insert*: *set* (*prefixes* (*w′* @ [*xy*])) = *Set.insert* (*w′*@[*xy*]) (*set* (*prefixes* (*w′*)))
⟨*proof*⟩

lemma *prefixes-set-subset*:
set (*prefixes xs*) ⊆ *set* (*prefixes* (*xs*@*ys*))
⟨*proof*⟩

lemma *prefixes-prefix-subset* :
assumes *xs* ∈ *set* (*prefixes ys*)
shows *set* (*prefixes xs*) ⊆ *set* (*prefixes ys*)
⟨*proof*⟩

lemma *prefixes-butlast-is-prefix* :
butlast xs ∈ *set* (*prefixes xs*)
⟨*proof*⟩

lemma *prefixes-take-iff* :
xs ∈ *set* (*prefixes ys*) ⟷ *take* (*length xs*) *ys* = *xs*
⟨*proof*⟩

lemma *prefixes-set-Nil* : [] ∈ *list.set* (*prefixes xs*)
⟨*proof*⟩

lemma *prefixes-prefixes* :
assumes *ys* ∈ *list.set* (*prefixes xs*)
zs ∈ *list.set* (*prefixes xs*)
shows *ys* ∈ *list.set* (*prefixes zs*) ∨ *zs* ∈ *list.set* (*prefixes ys*)
⟨*proof*⟩

1.6.1 Pairs of Distinct Prefixes

fun *prefix-pairs* :: *'a list* ⇒ (*'a list* × *'a list*) *list*
where *prefix-pairs* [] = [] |
prefix-pairs xs = *prefix-pairs* (*butlast xs*) @ (*map* (λ *ys*. (*ys*,*xs*)) (*butlast* (*prefixes xs*)))

lemma *prefixes-butlast* :
set (*butlast* (*prefixes xs*)) = {*ys* . ∃ *zs* . *ys*@*zs* = *xs* ∧ *zs* ≠ []}
⟨*proof*⟩

lemma *prefix-pairs-set* :
set (*prefix-pairs xs*) = {(*zs*,*ys*) | *zs ys* . ∃ *xs1 xs2* . *zs*@*xs1* = *ys* ∧ *ys*@*xs2* = *xs* ∧ *xs1* ≠ []}
⟨*proof*⟩

lemma *prefix-pairs-set-alt* :
set (*prefix-pairs xs*) = {(*xs1*,*xs1*@*xs2*) | *xs1 xs2* . *xs2* ≠ [] ∧ (∃ *xs3* . *xs1*@*xs2*@*xs3* = *xs*)}
⟨*proof*⟩

lemma *prefixes-Cons* :
assumes (*x*#*xs*) ∈ *set* (*prefixes* (*y*#*ys*))
shows *x* = *y* **and** *xs* ∈ *set* (*prefixes ys*)
⟨*proof*⟩

lemma *prefixes-prepend* :
assumes *xs'* ∈ *set* (*prefixes xs*)
shows *ys*@*xs'* ∈ *set* (*prefixes* (*ys*@*xs*))
⟨*proof*⟩

lemma *prefixes-prefix-suffix-ob* :
assumes *a* ∈ *set* (*prefixes* (*b*@*c*))

and $a \notin$ *set* (*prefixes* b)
obtains c' c'' **where** $c = c'@c''$
and $a = b@c'$
and $c' \neq []$
⟨*proof*⟩

fun *list-ordered-pairs* :: *'a list* ⇒ (*'a* × *'a*) *list* **where**
list-ordered-pairs [] = [] |
list-ordered-pairs (*x#xs*) = (*map* (*Pair x*) *xs*) @ (*list-ordered-pairs xs*)

lemma *list-ordered-pairs-set-containment* :
assumes $x \in$ *list.set xs*
and $y \in$ *list.set xs*
and $x \neq y$
shows $(x,y) \in$ *list.set* (*list-ordered-pairs xs*) ∨ $(y,x) \in$ *list.set* (*list-ordered-pairs xs*)
⟨*proof*⟩

1.7 Calculating Distinct Non-Reflexive Pairs over List Elements

fun *non-sym-dist-pairs'* :: *'a list* ⇒ (*'a* × *'a*) *list* **where**
non-sym-dist-pairs' [] = [] |
non-sym-dist-pairs' (*x#xs*) = (*map* (λ *y*. (*x*,*y*)) *xs*) @ *non-sym-dist-pairs' xs*

fun *non-sym-dist-pairs* :: *'a list* ⇒ (*'a* × *'a*) *list* **where**
non-sym-dist-pairs xs = *non-sym-dist-pairs'* (*remdups xs*)

lemma *non-sym-dist-pairs-subset* : *set* (*non-sym-dist-pairs xs*) ⊆ (*set xs*) × (*set xs*)
⟨*proof*⟩

lemma *non-sym-dist-pairs'-elems-distinct*:
assumes *distinct xs*
and $(x,y) \in$ *set* (*non-sym-dist-pairs' xs*)
shows $x \in$ *set xs*
and $y \in$ *set xs*
and $x \neq y$
⟨*proof*⟩

lemma *non-sym-dist-pairs-elems-distinct*:
assumes $(x,y) \in$ *set* (*non-sym-dist-pairs xs*)
shows $x \in$ *set xs*
and $y \in$ *set xs*
and $x \neq y$
⟨*proof*⟩

lemma *non-sym-dist-pairs-elems* :
  assumes $x \in$ *set xs*
  and $y \in$ *set xs*
  and $x \neq y$
shows $(x,y) \in$ *set* (*non-sym-dist-pairs xs*) $\vee$ $(y,x) \in$ *set* (*non-sym-dist-pairs xs*)
  ⟨*proof*⟩

lemma *non-sym-dist-pairs′-elems-non-refl* :
  assumes *distinct xs*
  and $(x,y) \in$ *set* (*non-sym-dist-pairs′ xs*)
shows $(y,x) \notin$ *set* (*non-sym-dist-pairs′ xs*)
  ⟨*proof*⟩

lemma *non-sym-dist-pairs-elems-non-refl* :
  assumes $(x,y) \in$ *set* (*non-sym-dist-pairs xs*)
  shows $(y,x) \notin$ *set* (*non-sym-dist-pairs xs*)
  ⟨*proof*⟩

lemma *non-sym-dist-pairs-set-iff* :
  $(x,y) \in$ *set* (*non-sym-dist-pairs xs*)
    $\longleftrightarrow$ ($x \neq y \wedge x \in$ *set xs* $\wedge$ $y \in$ *set xs* $\wedge$ $(y,x) \notin$ *set* (*non-sym-dist-pairs xs*))
  ⟨*proof*⟩

1.8 Finite Linear Order From List Positions

fun *linear-order-from-list-position′* :: *′a list* ⇒ (*′a* × *′a*) *list* **where**
  linear-order-from-list-position′ [] = [] |
  linear-order-from-list-position′ (*x#xs*)
    = (*x*,*x*) # (*map* (λ *y* . (*x*,*y*)) *xs*) @ (*linear-order-from-list-position′ xs*)

fun *linear-order-from-list-position* :: *′a list* ⇒ (*′a* × *′a*) *list* **where**
  linear-order-from-list-position xs = *linear-order-from-list-position′* (*remdups xs*)

lemma *linear-order-from-list-position-set* :
  set (*linear-order-from-list-position xs*)
    = (*set* (*map* (λ *x* . (*x*,*x*)) *xs*)) ∪ *set* (*non-sym-dist-pairs xs*)
  ⟨*proof*⟩

lemma *linear-order-from-list-position-total*:
  total-on (*set xs*) (*set* (*linear-order-from-list-position xs*))
  ⟨*proof*⟩

lemma *linear-order-from-list-position-refl*:

refl-on (*set xs*) (*set* (*linear-order-from-list-position xs*))
⟨*proof*⟩

lemma *linear-order-from-list-position-antisym*:
antisym (*set* (*linear-order-from-list-position xs*))
⟨*proof*⟩

lemma *non-sym-dist-pairs′-indices* :
distinct xs ⟹ (*x*,*y*) ∈ *set* (*non-sym-dist-pairs′ xs*)
⟹ (∃ *i j* . *xs* ! *i* = *x* ∧ *xs* ! *j* = *y* ∧ *i* < *j* ∧ *i* < *length xs* ∧ *j* < *length xs*)
⟨*proof*⟩

lemma *non-sym-dist-pairs′-trans*: *distinct xs* ⟹ *trans* (*set* (*non-sym-dist-pairs′ xs*))
⟨*proof*⟩

lemma *non-sym-dist-pairs-trans*: *trans* (*set* (*non-sym-dist-pairs xs*))
⟨*proof*⟩

lemma *linear-order-from-list-position-trans*: *trans* (*set* (*linear-order-from-list-position xs*))
⟨*proof*⟩

1.9 Find And Remove in a Single Pass

fun *find-remove′* :: (′*a* ⇒ *bool*) ⇒ ′*a list* ⇒ ′*a list* ⇒ (′*a* × ′*a list*) *option* **where**
find-remove′ P [] - = *None* |
find-remove′ P (*x*#*xs*) *prev* = (*if P x*
then Some (*x*,*prev*@*xs*)
else find-remove′ P xs (*prev*@[*x*]))

fun *find-remove* :: (′*a* ⇒ *bool*) ⇒ ′*a list* ⇒ (′*a* × ′*a list*) *option* **where**
find-remove P xs = *find-remove′ P xs* []

lemma *find-remove′-set* :
assumes *find-remove′ P xs prev* = *Some* (*x*,*xs′*)
shows *P x*
and *x* ∈ *set xs*
and *xs′* = *prev*@(*remove1 x xs*)
⟨*proof*⟩

lemma *find-remove′-set-rev* :
assumes *x* ∈ *set xs*

and $P\ x$
shows *find-remove′ P xs prev* $\neq$ *None*
⟨*proof*⟩

lemma *find-remove-None-iff* :
find-remove P xs = *None* $\longleftrightarrow \neg (\exists x\ .\ x \in$ *set xs* $\wedge\ P\ x)$
⟨*proof*⟩

lemma *find-remove-set* :
assumes *find-remove P xs* = *Some* (*x*,*xs′*)
shows *P x*
and $x \in$ *set xs*
and *xs′* = (*remove1 x xs*)
⟨*proof*⟩

fun *find-remove-2′* :: (′*a*⇒′*b*⇒*bool*) ⇒ ′*a list* ⇒ ′*b list* ⇒ ′*a list* ⇒ (′*a* × ′*b* × ′*a list*) *option*
where
find-remove-2′ P [] - - = *None* |
find-remove-2′ P (*x*#*xs*) *ys prev* = (*case find* (λ*y* . *P x y*) *ys of*
Some y ⇒ *Some* (*x*,*y*,*prev*@*xs*) |
None ⇒ *find-remove-2′ P xs ys* (*prev*@[*x*]))

fun *find-remove-2* :: (′*a* ⇒ ′*b* ⇒ *bool*) ⇒ ′*a list* ⇒ ′*b list* ⇒ (′*a* × ′*b* × ′*a list*) *option* **where**
find-remove-2 P xs ys = *find-remove-2′ P xs ys* []

lemma *find-remove-2′-set* :
assumes *find-remove-2′ P xs ys prev* = *Some* (*x*,*y*,*xs′*)
shows *P x y*
and $x \in$ *set xs*
and $y \in$ *set ys*
and *distinct* (*prev*@*xs*) ⟹ *set xs′* = (*set prev* ∪ *set xs*) − {*x*}
and *distinct* (*prev*@*xs*) ⟹ *distinct xs′*
and *xs′* = *prev*@(*remove1 x xs*)
and *find* (*P x*) *ys* = *Some y*
⟨*proof*⟩

lemma *find-remove-2′-strengthening* :
assumes *find-remove-2′ P xs ys prev* = *Some* (*x*,*y*,*xs′*)
and $P'\ x\ y$
and $\bigwedge x'\ y'\ .\ P'\ x'\ y' \Longrightarrow P\ x'\ y'$

shows *find-remove-2′ P′ xs ys prev* = *Some* (*x*,*y*,*xs′*)
⟨*proof*⟩

lemma *find-remove-2-strengthening* :
assumes *find-remove-2 P xs ys* = *Some* (*x*,*y*,*xs′*)
and $P'\ x\ y$
and $\bigwedge x'\ y'\ .\ P'\ x'\ y' \Longrightarrow P\ x'\ y'$
shows *find-remove-2 P′ xs ys* = *Some* (*x*,*y*,*xs′*)
⟨*proof*⟩

lemma *find-remove-2′-prev-independence* :
assumes *find-remove-2′ P xs ys prev* = *Some* (*x*,*y*,*xs′*)
shows ∃ *xs′′* . *find-remove-2′ P xs ys prev′* = *Some* (*x*,*y*,*xs′′*)
⟨*proof*⟩

lemma *find-remove-2′-filter* :
assumes *find-remove-2′ P* (*filter P′ xs*) *ys prev* = *Some* (*x*,*y*,*xs′*)
and $\bigwedge x\ y\ .\ \neg\ P'\ x \Longrightarrow \neg\ P\ x\ y$
shows ∃ *xs′′* . *find-remove-2′ P xs ys prev* = *Some* (*x*,*y*,*xs′′*)
⟨*proof*⟩

lemma *find-remove-2-filter* :
assumes *find-remove-2 P* (*filter P′ xs*) *ys* = *Some* (*x*,*y*,*xs′*)
and $\bigwedge x\ y\ .\ \neg\ P'\ x \Longrightarrow \neg\ P\ x\ y$
shows ∃ *xs′′* . *find-remove-2 P xs ys* = *Some* (*x*,*y*,*xs′′*)
⟨*proof*⟩

lemma *find-remove-2′-index* :
assumes *find-remove-2′ P xs ys prev* = *Some* (*x*,*y*,*xs′*)
obtains *i i′* **where** *i* < *length xs*
xs ! *i* = *x*
$\bigwedge j\ .\ j < i \Longrightarrow$ *find* (λ*y* . *P* (*xs* ! *j*) *y*) *ys* = *None*
i′ < *length ys*
ys ! *i′* = *y*
$\bigwedge j\ .\ j < i' \Longrightarrow \neg$ *P* (*xs* ! *i*) (*ys* ! *j*)
⟨*proof*⟩

lemma *find-remove-2-index* :
assumes *find-remove-2 P xs ys* = *Some* (*x*,*y*,*xs′*)
obtains *i i′* **where** *i* < *length xs*
xs ! *i* = *x*
$\bigwedge j\ .\ j < i \Longrightarrow$ *find* (λ*y* . *P* (*xs* ! *j*) *y*) *ys* = *None*
i′ < *length ys*
ys ! *i′* = *y*

$\bigwedge j . j < i' \Longrightarrow \neg P (xs ! i) (ys ! j)$

⟨*proof*⟩

lemma *find-remove-2'-set-rev* :
assumes $x \in set\ xs$
and $y \in set\ ys$
and $P\ x\ y$
shows *find-remove-2' P xs ys prev* $\neq$ *None*
⟨*proof*⟩

lemma *find-remove-2'-diff-prev-None* :
(*find-remove-2' P xs ys prev* = *None* $\Longrightarrow$ *find-remove-2' P xs ys prev'* = *None*)
⟨*proof*⟩

lemma *find-remove-2'-diff-prev-Some* :
(*find-remove-2' P xs ys prev* = *Some* (x,y,xs')
$\Longrightarrow \exists\ xs''$. *find-remove-2' P xs ys prev'* = *Some* (x,y,xs''))
⟨*proof*⟩

lemma *find-remove-2-None-iff* :
find-remove-2 P xs ys = *None* $\longleftrightarrow \neg (\exists x\ y . x \in set\ xs \wedge y \in set\ ys \wedge P\ x\ y)$
⟨*proof*⟩

lemma *find-remove-2-set* :
assumes *find-remove-2 P xs ys* = *Some* (x,y,xs')
shows $P\ x\ y$
and $x \in set\ xs$
and $y \in set\ ys$
and *distinct xs* $\Longrightarrow$ *set xs'* = $(set\ xs) - \{x\}$
and *distinct xs* $\Longrightarrow$ *distinct xs'*
and xs' = (*remove1 x xs*)
⟨*proof*⟩

lemma *find-remove-2-removeAll* :
assumes *find-remove-2 P xs ys* = *Some* (x,y,xs')
and *distinct xs*
shows xs' = *removeAll x xs*
⟨*proof*⟩

lemma *find-remove-2-length* :
assumes *find-remove-2 P xs ys* = *Some* (x,y,xs')
shows *length xs'* = *length xs* − *1*
⟨*proof*⟩

fun *separate-by* :: (′*a* ⇒ *bool*) ⇒ ′*a list* ⇒ (′*a list* × ′*a list*) **where**
separate-by P xs = (*filter P xs*, *filter* (λ *x* . ¬ *P x*) *xs*)

lemma *separate-by-code*[*code*] :
separate-by P xs = *foldr* (λ*x* (*prevPass*,*prevFail*) . *if P x then* (*x*#*prevPass*,*prevFail*) *else* (*prevPass*,*x*#*prevFail*)) *xs* ([],[])
⟨*proof*⟩

fun *find-remove-2-all* :: (′*a* ⇒ ′*b* ⇒ *bool*) ⇒ ′*a list* ⇒ ′*b list* ⇒ ((′*a* × ′*b*) *list* × ′*a list*) **where**
find-remove-2-all P xs ys =
(*map* (λ *x* . (*x*, *the* (*find* (λ*y* . *P x y*) *ys*))) (*filter* (λ *x* . *find* (λ*y* . *P x y*) *ys* ≠ *None*) *xs*)
,*filter* (λ *x* . *find* (λ*y* . *P x y*) *ys* = *None*) *xs*)

fun *find-remove-2-all*′ :: (′*a* ⇒ ′*b* ⇒ *bool*) ⇒ ′*a list* ⇒ ′*b list* ⇒ ((′*a* × ′*b*) *list* × ′*a list*) **where**
find-remove-2-all′ *P xs ys* =
(*let* (*successesWithWitnesses*,*failures*) = *separate-by* (λ(*x*,*y*) . *y* ≠ *None*) (*map* (λ *x* . (*x*,*find* (λ*y* . *P x y*) *ys*)) *xs*)
in (*map* (λ (*x*,*y*) . (*x*, *the y*)) *successesWithWitnesses*, *map fst failures*))

lemma *find-remove-2-all-code*[*code*] :
find-remove-2-all P xs ys = *find-remove-2-all*′ *P xs ys*
⟨*proof*⟩

1.10 Set-Operations on Lists

fun *pow-list* :: ′*a list* ⇒ ′*a list list* **where**
pow-list [] = [[]] |
pow-list (*x*#*xs*) = (*let pxs* = *pow-list xs in pxs* @ *map* (λ *ys* . *x*#*ys*) *pxs*)

lemma *pow-list-set* :
set (*map set* (*pow-list xs*)) = *Pow* (*set xs*)
⟨*proof*⟩

fun *inter-list* :: ′*a list* ⇒ ′*a list* ⇒ ′*a list* **where**
inter-list xs ys = *filter* (λ *x* . *x* ∈ *set ys*) *xs*

lemma *inter-list-set* : *set* (*inter-list xs ys*) = (*set xs*) ∩ (*set ys*)
⟨*proof*⟩

fun *subset-list* :: ′*a list* ⇒ ′*a list* ⇒ *bool* **where**
subset-list xs ys = *list-all* (λ *x* . *x* ∈ *set ys*) *xs*

lemma *subset-list-set* : *subset-list xs ys* = ((*set xs*) ⊆ (*set ys*))

⟨*proof*⟩

1.10.1 Removing Subsets in a List of Sets

lemma *remove1-length* : $x \in set\ xs \Longrightarrow length\ (remove1\ x\ xs) < length\ xs$
⟨*proof*⟩

function *remove-subsets* :: $'a\ set\ list \Rightarrow 'a\ set\ list$ **where**
remove-subsets [] = [] |
remove-subsets $(x\#xs) = (case\ find\text{-}remove\ (\lambda\ y\ .\ x \subset y)\ xs\ of$
$Some\ (y',xs') \Rightarrow remove\text{-}subsets\ (y'\#\ (filter\ (\lambda\ y\ .\ \neg(y \subseteq x))\ xs'))$ |
$None \Rightarrow x\ \#\ (remove\text{-}subsets\ (filter\ (\lambda\ y\ .\ \neg(y \subseteq x))\ xs)))$
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *remove-subsets-set* : $set\ (remove\text{-}subsets\ xss) = \{xs\ .\ xs \in set\ xss \wedge (\nexists\ xs'\ .\ xs' \in set\ xss \wedge xs \subset xs')\}$
⟨*proof*⟩

1.11 Linear Order on Sum

instantiation *sum* :: (*ord*,*ord*) *ord*
begin

fun *less-eq-sum* :: $'a + 'b \Rightarrow 'a + 'b \Rightarrow bool$ **where**
less-eq-sum (*Inl a*) (*Inl b*) = $(a \leq b)$ |
less-eq-sum (*Inl a*) (*Inr b*) = *True* |
less-eq-sum (*Inr a*) (*Inl b*) = *False* |
less-eq-sum (*Inr a*) (*Inr b*) = $(a \leq b)$

fun *less-sum* :: $'a + 'b \Rightarrow 'a + 'b \Rightarrow bool$ **where**
less-sum $a\ b = (a \leq b \wedge a \neq b)$

instance ⟨*proof*⟩
end

instantiation *sum* :: (*linorder*,*linorder*) *linorder*
begin

lemma *less-le-not-le-sum* :
fixes $x :: 'a + 'b$
and $y :: 'a + 'b$
shows $(x < y) = (x \leq y \wedge \neg\ y \leq x)$
⟨*proof*⟩

lemma *order-refl-sum* :

fixes $x :: 'a + 'b$
shows $x \leq x$
⟨*proof*⟩

lemma *order-trans-sum* :
fixes $x :: 'a + 'b$
fixes $y :: 'a + 'b$
fixes $z :: 'a + 'b$
shows $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$
⟨*proof*⟩

lemma *antisym-sum* :
fixes $x :: 'a + 'b$
fixes $y :: 'a + 'b$
shows $x \leq y \Longrightarrow y \leq x \Longrightarrow x = y$
⟨*proof*⟩

lemma *linear-sum* :
fixes $x :: 'a + 'b$
fixes $y :: 'a + 'b$
shows $x \leq y \vee y \leq x$
⟨*proof*⟩

instance
⟨*proof*⟩
end

1.12 Removing Proper Prefixes

definition *remove-proper-prefixes* :: *'a list set* ⇒ *'a list set* **where**
remove-proper-prefixes xs $= \{x \ . \ x \in xs \wedge (\nexists \ x' \ . \ x' \neq [] \wedge x@x' \in xs)\}$

lemma *remove-proper-prefixes-code*[*code*] :
remove-proper-prefixes (*set xs*) = *set* (*filter* (λx . ($\forall$ $y \in$ *set xs* . *is-prefix x y* $\longrightarrow$ $x = y$)) *xs*)
⟨*proof*⟩

1.13 Underspecified List Representations of Sets

definition *as-list-helper* :: *'a set* ⇒ *'a list* **where**
as-list-helper X = (*SOME xs* . *set xs* = $X \wedge$ *distinct xs*)

lemma *as-list-helper-props* :
assumes *finite X*
shows *set* (*as-list-helper X*) = *X*
and *distinct* (*as-list-helper X*)
⟨*proof*⟩

1.14 Assigning indices to elements of a finite set

fun *assign-indices* :: ('*a* :: *linorder*) *set* ⇒ ('*a* ⇒ *nat*) **where**
assign-indices xs = (λ *x* . *the* (*find-index* ((=)*x*) (*sorted-list-of-set xs*)))

lemma *assign-indices-bij*:
assumes *finite xs*
shows *bij-betw* (*assign-indices xs*) *xs* {..<*card xs*}
⟨*proof*⟩

1.15 Other Lemmata

lemma *foldr-elem-check*:
assumes *list.set xs* ⊆ *A*
shows *foldr* (λ *x y* . *if x* ∉ *A then y else f x y*) *xs v* = *foldr f xs v*
⟨*proof*⟩

lemma *foldl-elem-check*:
assumes *list.set xs* ⊆ *A*
shows *foldl* (λ *y x* . *if x* ∉ *A then y else f y x*) *v xs* = *foldl f v xs*
⟨*proof*⟩

lemma *foldr-length-helper* :
assumes *length xs* = *length ys*
shows *foldr* (λ- *x* . *f x*) *xs b* = *foldr* (λ*a x* . *f x*) *ys b*
⟨*proof*⟩

lemma *list-append-subset3* : *set xs1* ⊆ *set ys1* ⟹ *set xs2* ⊆ *set ys2* ⟹ *set xs3* ⊆ *set ys3* ⟹ *set* (*xs1*@*xs2*@*xs3*) ⊆ *set*(*ys1*@*ys2*@*ys3*) ⟨*proof*⟩

lemma *subset-filter* : *set xs* ⊆ *set ys* ⟹ *set xs* = *set* (*filter* (λ *x* . *x* ∈ *set xs*) *ys*)
⟨*proof*⟩

lemma *map-filter-elem* :
assumes *y* ∈ *set* (*List.map-filter f xs*)
obtains *x* **where** *x* ∈ *set xs*
and *f x* = *Some y*
⟨*proof*⟩

lemma *filter-length-weakening* :
assumes ⋀ *q* . *f1 q* ⟹ *f2 q*
shows *length* (*filter f1 p*) ≤ *length* (*filter f2 p*)
⟨*proof*⟩

lemma *max-length-elem* :
fixes *xs* :: '*a list set*
assumes *finite xs*
and *xs* ≠ {}
shows ∃ *x* ∈ *xs* . ¬(∃ *y* ∈ *xs* . *length y* > *length x*)
⟨*proof*⟩

lemma *min-length-elem* :
  fixes *xs* :: *'a list set*
  assumes *finite xs*
  and $xs \neq \{\}$
shows $\exists\ x \in xs\ .\ \neg(\exists\ y \in xs\ .\ length\ y < length\ x)$
⟨*proof*⟩

lemma *list-property-from-index-property* :
  assumes $\bigwedge\ i\ .\ i < length\ xs \Longrightarrow P\ (xs\ !\ i)$
  shows $\bigwedge\ x\ .\ x \in set\ xs \Longrightarrow P\ x$
  ⟨*proof*⟩

lemma *list-distinct-prefix* :
  assumes $\bigwedge\ i\ .\ i < length\ xs \Longrightarrow xs\ !\ i \notin set\ (take\ i\ xs)$
  shows *distinct xs*
⟨*proof*⟩

lemma *concat-pair-set* :
  $set\ (concat\ (map\ (\lambda x.\ map\ (Pair\ x)\ ys)\ xs)) = \{xy\ .\ fst\ xy \in set\ xs \wedge snd\ xy \in set\ ys\}$
  ⟨*proof*⟩

lemma *list-set-sym* :
  $set\ (x@y) = set\ (y@x)$ ⟨*proof*⟩

lemma *list-contains-last-take* :
  assumes $x \in set\ xs$
  shows $\exists\ i\ .\ 0 < i \wedge i \leq length\ xs \wedge last\ (take\ i\ xs) = x$
  ⟨*proof*⟩

lemma *take-last-index* :
  assumes $i < length\ xs$
  shows $last\ (take\ (Suc\ i)\ xs) = xs\ !\ i$
  ⟨*proof*⟩

lemma *integer-singleton-least* :
  assumes $\{x\ .\ P\ x\} = \{a::integer\}$
  shows $a = (LEAST\ x\ .\ P\ x)$
  ⟨*proof*⟩

lemma *sort-list-split* :
  $\forall\ x \in set\ (take\ i\ (sort\ xs))\ .\ \forall\ y \in set\ (drop\ i\ (sort\ xs))\ .\ x \leq y$
  ⟨*proof*⟩

lemma *set-map-subset* :
assumes $x \in set\ xs$
and $t \in set\ (map\ f\ [x])$
shows $t \in set\ (map\ f\ xs)$
⟨*proof*⟩

lemma *rev-induct2*[*consumes 1*, *case-names Nil snoc*]:
assumes $length\ xs = length\ ys$
and $P\ []\ []$
and $(\bigwedge x\ xs\ y\ ys.\ length\ xs = length\ ys \Longrightarrow P\ xs\ ys \Longrightarrow P\ (xs@[x])\ (ys@[y]))$
shows $P\ xs\ ys$
⟨*proof*⟩

lemma *finite-set-min-param-ex* :
assumes $finite\ XS$
and $\bigwedge\ x\ .\ x \in XS \Longrightarrow \exists\ k\ .\ \forall\ k'\ .\ k \leq k' \longrightarrow P\ x\ k'$
shows $\exists\ (k::nat)\ .\ \forall\ x \in XS\ .\ P\ x\ k$
⟨*proof*⟩

fun *list-max* :: *nat list* ⇒ *nat* **where**
list-max [] = *0* |
list-max xs = *Max* (*set xs*)

lemma *list-max-is-max* : $q \in set\ xs \Longrightarrow q \leq list\text{-}max\ xs$
⟨*proof*⟩

lemma *list-prefix-subset* : $\exists\ ys\ .\ ts = xs@ys \Longrightarrow set\ xs \subseteq set\ ts$ ⟨*proof*⟩
lemma *list-map-set-prop* : $x \in set\ (map\ f\ xs) \Longrightarrow \forall\ y\ .\ P\ (f\ y) \Longrightarrow P\ x$ ⟨*proof*⟩
lemma *list-concat-non-elem* : $x \notin set\ xs \Longrightarrow x \notin set\ ys \Longrightarrow x \notin set\ (xs@ys)$ ⟨*proof*⟩
lemma *list-prefix-elem* : $x \in set\ (xs@ys) \Longrightarrow x \notin set\ ys \Longrightarrow x \in set\ xs$ ⟨*proof*⟩
lemma *list-map-source-elem* : $x \in set\ (map\ f\ xs) \Longrightarrow \exists\ x' \in set\ xs\ .\ x = f\ x'$
⟨*proof*⟩

lemma *maximal-set-cover* :
fixes X :: *'a set set*
assumes $finite\ X$
and $S \in X$
shows $\exists\ S' \in X\ .\ S \subseteq S' \land (\forall\ S'' \in X\ .\ \neg(S' \subset S''))$
⟨*proof*⟩

lemma *map-set* :
assumes $x \in set\ xs$
shows $f\ x \in set\ (map\ f\ xs)$ ⟨*proof*⟩

lemma *maximal-distinct-prefix* :
assumes $\neg$ *distinct xs*
obtains *n* **where** *distinct* (*take* (*Suc n*) *xs*)
and $\neg$ (*distinct* (*take* (*Suc* (*Suc n*)) *xs*))
⟨*proof*⟩

lemma *distinct-not-in-prefix* :
assumes $\bigwedge$ *i* . ($\bigwedge$ *x* . *x* $\in$ *set* (*take i xs*) $\Longrightarrow$ *xs* ! *i* $\neq$ *x*)
shows *distinct xs*
⟨*proof*⟩

lemma *list-index-fun-gt* : $\bigwedge$ *xs* (*f*::'*a* $\Rightarrow$ *nat*) *i j* .
($\bigwedge$ *i* . *Suc i* < *length xs* $\Longrightarrow$ *f* (*xs* ! *i*) > *f* (*xs* ! (*Suc i*)))
$\Longrightarrow$ *j* < *i*
$\Longrightarrow$ *i* < *length xs*
$\Longrightarrow$ *f* (*xs* ! *j*) > *f* (*xs* ! *i*)
⟨*proof*⟩

lemma *finite-set-elem-maximal-extension-ex* :
assumes *xs* $\in$ *S*
and *finite S*
shows $\exists$ *ys* . *xs*@*ys* $\in$ *S* $\wedge$ $\neg$ ($\exists$ *zs* . *zs* $\neq$ [] $\wedge$ *xs*@*ys*@*zs* $\in$ *S*)
⟨*proof*⟩

lemma *list-index-split-set*:
assumes *i* < *length xs*
shows *set xs* = *set* ((*xs* ! *i*) # ((*take i xs*) @ (*drop* (*Suc i*) *xs*)))
⟨*proof*⟩

lemma *max-by-foldr* :
assumes *x* $\in$ *set xs*
shows *f x* < *Suc* (*foldr* (λ *x'* *m* . *max* (*f x'*) *m*) *xs 0*)
⟨*proof*⟩

lemma *Max-elem* : *finite* (*xs* :: '*a set*) $\Longrightarrow$ *xs* $\neq$ {} $\Longrightarrow$ $\exists$ *x* $\in$ *xs* . *Max* (*image* (*f* :: '*a* $\Rightarrow$ *nat*) *xs*) = *f x*
⟨*proof*⟩

lemma *card-union-of-singletons* :
assumes $\bigwedge$ *S* . *S* $\in$ *SS* $\Longrightarrow$ ($\exists$ *t* . *S* = {*t*})
shows *card* ($\bigcup$ *SS*) = *card SS*
⟨*proof*⟩

lemma *card-union-of-distinct* :

assumes $\bigwedge$ *S1 S2* . *S1* ∈ *SS* ⟹ *S2* ∈ *SS* ⟹ *S1* = *S2* ∨ *f S1* ∩ *f S2* = {}
and *finite SS*
and $\bigwedge$ *S* . *S* ∈ *SS* ⟹ *f S* ≠ {}
shows *card* (*image f SS*) = *card SS*
⟨*proof*⟩

lemma *take-le* :
assumes *i* ≤ *length xs*
shows *take i* (*xs*@*ys*) = *take i xs*
⟨*proof*⟩

lemma *butlast-take-le* :
assumes *i* ≤ *length* (*butlast xs*)
shows *take i* (*butlast xs*) = *take i xs*
⟨*proof*⟩

lemma *distinct-union-union-card* :
assumes *finite xs*
and $\bigwedge$ *x1 x2 y1 y2* . *x1* ≠ *x2* ⟹ *x1* ∈ *xs* ⟹ *x2* ∈ *xs* ⟹ *y1* ∈ *f x1* ⟹ *y2* ∈ *f x2* ⟹ *g y1* ∩ *g y2* = {}
and $\bigwedge$ *x1 y1 y2* . *y1* ∈ *f x1* ⟹ *y2* ∈ *f x1* ⟹ *y1* ≠ *y2* ⟹ *g y1* ∩ *g y2* = {}
and $\bigwedge$ *x1* . *finite* (*f x1*)
and $\bigwedge$ *y1* . *finite* (*g y1*)
and $\bigwedge$ *y1* . *g y1* ⊆ *zs*
and *finite zs*
shows ($\sum$ *x* ∈ *xs* . *card* ($\bigcup$ *y* ∈ *f x* . *g y*)) ≤ *card zs*
⟨*proof*⟩

lemma *set-concat-elem* :
assumes *x* ∈ *set* (*concat xss*)
obtains *xs* **where** *xs* ∈ *set xss* **and** *x* ∈ *set xs*
⟨*proof*⟩

lemma *set-map-elem* :
assumes *y* ∈ *set* (*map f xs*)
obtains *x* **where** *y* = *f x* **and** *x* ∈ *set xs*
⟨*proof*⟩

lemma *finite-snd-helper*:
assumes *finite xs*
shows *finite* {*z*. ((*q*, *p*), *z*) ∈ *xs*}
⟨*proof*⟩

lemma *fold-dual* : *fold* (λ *x* (*a1*,*a2*) . (*g1 x a1*, *g2 x a2*)) *xs* (*a1*,*a2*) = (*fold g1*

xs a1, *fold g2 xs a2*)
⟨*proof*⟩

lemma *recursion-renaming-helper* :
assumes $f1 = (\lambda x . \text{ if } P\ x \text{ then } x \text{ else } f1\ (Suc\ x))$
and $f2 = (\lambda x . \text{ if } P\ x \text{ then } x \text{ else } f2\ (Suc\ x))$
and $\bigwedge x . x \geq k \Longrightarrow P\ x$
shows $f1 = f2$
⟨*proof*⟩

lemma *minimal-fixpoint-helper* :
assumes $f = (\lambda x . \text{ if } P\ x \text{ then } x \text{ else } f\ (Suc\ x))$
and $\bigwedge x . x \geq k \Longrightarrow P\ x$
shows $P\ (f\ x)$
and $\bigwedge x' . x' \geq x \Longrightarrow x' < f\ x \Longrightarrow \neg\ P\ x'$
⟨*proof*⟩

lemma *map-set-index-helper* :
assumes $xs \neq []$
shows $set\ (map\ f\ xs) = (\lambda i . f\ (xs\ !\ i))\ `\ \{..\ (length\ xs - 1)\}$
⟨*proof*⟩

lemma *partition-helper* :
assumes *finite X*
and $X \neq \{\}$
and $\bigwedge x . x \in X \Longrightarrow p\ x \subseteq X$
and $\bigwedge x . x \in X \Longrightarrow p\ x \neq \{\}$
and $\bigwedge x\ y . x \in X \Longrightarrow y \in X \Longrightarrow p\ x = p\ y \vee p\ x \cap p\ y = \{\}$
and $(\bigcup x \in X . p\ x) = X$
obtains $l{::}nat$ **and** p' **where**
$p'\ `\ \{..l\} = p\ `\ X$
$\bigwedge i\ j . i \leq l \Longrightarrow j \leq l \Longrightarrow i \neq j \Longrightarrow p'\ i \cap p'\ j = \{\}$
$card\ (p\ `\ X) = Suc\ l$
⟨*proof*⟩

lemma *take-diff* :
assumes $i \leq length\ xs$
and $j \leq length\ xs$
and $i \neq j$
shows $take\ i\ xs \neq take\ j\ xs$
⟨*proof*⟩

lemma *image-inj-card-helper* :
assumes *finite X*
and $\bigwedge a\ b . a \in X \Longrightarrow b \in X \Longrightarrow a \neq b \Longrightarrow f\ a \neq f\ b$
shows $card\ (f\ `\ X) = card\ X$
⟨*proof*⟩

lemma *sum-image-inj-card-helper* :
fixes l :: *nat*
assumes $\bigwedge i .\ i \leq l \Longrightarrow$ *finite* $(I\ i)$
and $\bigwedge i\ j .\ i \leq l \Longrightarrow j \leq l \Longrightarrow i \neq j \Longrightarrow I\ i \cap I\ j = \{\}$
shows $(\sum\ i \in \{..l\} .\ (card\ (I\ i))) = card\ (\bigcup\ i \in \{..l\} .\ I\ i)$
$\langle proof \rangle$

lemma *Min-elem* : *finite* $(xs :: {'a}\ set) \Longrightarrow xs \neq \{\} \Longrightarrow \exists\ x \in xs .\ Min\ (image\ (f :: {'a} \Rightarrow nat)\ xs) = f\ x$
$\langle proof \rangle$

lemma *finite-subset-mapping-limit* :
fixes $f :: nat \Rightarrow {'a}\ set$
assumes *finite* $(f\ 0)$
and $\bigwedge i\ j .\ i \leq j \Longrightarrow f\ j \subseteq f\ i$
obtains k **where** $\bigwedge k' .\ k \leq k' \Longrightarrow f\ k' = f\ k$
$\langle proof \rangle$

lemma *finite-card-less-witnesses* :
assumes *finite* A
and $card\ (g\ `\ A) < card\ (f\ `\ A)$
obtains $a\ b$ **where** $a \in A$ **and** $b \in A$ **and** $f\ a \neq f\ b$ **and** $g\ a = g\ b$
$\langle proof \rangle$

lemma *monotone-function-with-limit-witness-helper* :
fixes $f :: nat \Rightarrow nat$
assumes $\bigwedge i\ j .\ i \leq j \Longrightarrow f\ i \leq f\ j$
and $\bigwedge i\ j\ m .\ i < j \Longrightarrow f\ i = f\ j \Longrightarrow j \leq m \Longrightarrow f\ i = f\ m$
and $\bigwedge i .\ f\ i \leq k$
obtains x **where** $f\ (Suc\ x) = f\ x$ **and** $x \leq k - f\ 0$
$\langle proof \rangle$

lemma *different-lists-shared-prefix* :
assumes $xs \neq xs'$
obtains i **where** $take\ i\ xs = take\ i\ xs'$
and $take\ (Suc\ i)\ xs \neq take\ (Suc\ i)\ xs'$
$\langle proof \rangle$

lemma *foldr-funion-fempty* : $foldr\ (|\cup|)\ xs\ fempty = ffUnion\ (fset\text{-}of\text{-}list\ xs)$
$\langle proof \rangle$

lemma *foldr-funion-fsingleton* : $foldr\ (|\cup|)\ xs\ x = ffUnion\ (fset\text{-}of\text{-}list\ (x\#xs))$
$\langle proof \rangle$

lemma *foldl-funion-fempty* : $foldl\ (|\cup|)\ fempty\ xs = ffUnion\ (fset\text{-}of\text{-}list\ xs)$
$\langle proof \rangle$

lemma *foldl-funion-fsingleton* : $foldl\ (|\cup|)\ x\ xs = ffUnion\ (fset\text{-}of\text{-}list\ (x\#xs))$

⟨*proof*⟩

lemma *ffUnion-fmember-ob* : x |∈| *ffUnion XS* ⟹ ∃ *X* . *X* |∈| *XS* ∧ *x* |∈| *X*
⟨*proof*⟩

lemma *filter-not-all-length* :
filter P xs ≠ [] ⟹ *length* (*filter* (λ *x* . ¬ *P x*) *xs*) < *length xs*
⟨*proof*⟩

lemma *foldr-funion-fmember* : *B* |⊆| (*foldr* (|∪|) *A B*)
⟨*proof*⟩

lemma *prefix-free-set-maximal-list-ob* :
assumes *finite xs*
and $x \in xs$
obtains x' **where** $x@x' \in xs$ **and** ∄ y' . $y' \neq [] \wedge (x@x')@y' \in xs$
⟨*proof*⟩

lemma *map-upds-map-set-left* :
assumes [*map f xs* [↦] *xs*] *q* = *Some x*
shows *x* ∈ *set xs* **and** *q* = *f x*
⟨*proof*⟩

lemma *map-upds-map-set-right* :
assumes *x* ∈ *set xs*
shows [*xs* [↦] *map f xs*] *x* = *Some* (*f x*)
⟨*proof*⟩

lemma *map-upds-overwrite* :
assumes *x* ∈ *set xs*
and *length xs* = *length ys*
shows (*m*(*xs*[↦]*ys*)) *x* = [*xs*[↦]*ys*] *x*
⟨*proof*⟩

lemma *ran-dom-the-eq* : (λ*k* . *the* (*m k*)) ‘ *dom m* = *ran m*
⟨*proof*⟩

lemma *map-pair-fst* :
map fst (*map* (λ*x* . (*x*,*f x*)) *xs*) = *xs*
⟨*proof*⟩

lemma *map-of-map-pair-entry*: *map-of* (*map* (λ*k*. (*k*, *f k*)) *xs*) *x* = (*if x* ∈ *list.set xs then Some* (*f x*) *else None*)
⟨*proof*⟩

lemma *map-filter-alt-def* :
List.map-filter f1′ xs = *map the* (*filter* (λ*x* . *x* ≠ *None*) (*map f1′ xs*))
⟨*proof*⟩

lemma *map-filter-Nil* :
List.map-filter f1′ xs = [] ⟷ (∀ *x* ∈ *list.set xs* . *f1′ x* = *None*)
⟨*proof*⟩

lemma *sorted-list-of-set-set*: *set* ((*sorted-list-of-set* ∘ *set*) *xs*) = *set xs*
⟨*proof*⟩

fun *mapping-of* :: (′*a* × ′*b*) *list* ⇒ (′*a*, ′*b*) *mapping* **where**
mapping-of kvs = *foldl* (λ*m kv* . *Mapping.update* (*fst kv*) (*snd kv*) *m*) *Mapping.empty kvs*

lemma *mapping-of-map-of* :
assumes *distinct* (*map fst kvs*)
shows *Mapping.lookup* (*mapping-of kvs*) = *map-of kvs*
⟨*proof*⟩

lemma *map-pair-fst-helper* :
map fst (*map* (λ (*x1*,*x2*) . ((*x1*,*x2*), *f x1 x2*)) *xs*) = *xs*
⟨*proof*⟩

end

2 Refinements for Utilities

Introduces program refinement for *Util.thy*.

theory *Util-Refined*
imports *Util Containers.Containers*
begin

2.1 New Code Equations for *set-as-map*

declare [[*code drop*: *set-as-map*]]

lemma *set-as-map-refined*[*code*] :
fixes *t* :: (′*a* :: *ccompare* × ′*c* :: *ccompare*) *set-rbt*
and *xs*:: (′*b* :: *ceq* × ′*d* :: *ceq*) *set-dlist*
shows *set-as-map* (*RBT-set t*) = (*case ID CCOMPARE*((′*a* × ′*c*)) *of*
Some - ⇒ *Mapping.lookup* (*RBT-Set2.fold* (λ (*x*,*z*) *m* . *case Mapping.lookup m* (*x*) *of*
None ⇒ *Mapping.update* (*x*) {*z*} *m* |
Some zs ⇒ *Mapping.update* (*x*) (*Set.insert z zs*) *m*)
t
Mapping.empty) |

None ⇒ *Code.abort* (*STR ''set-as-map RBT-set*: *ccompare* = *None''*)
(λ-. *set-as-map* (*RBT-set t*)))
(**is** *?C1*)
and *set-as-map* (*DList-set xs*) = (*case ID CEQ*(('*b* × '*d*)) *of*
Some - ⇒ *Mapping.lookup* (*DList-Set.fold* (λ (*x*,*z*) *m* . *case Mapping.lookup m* (*x*) *of*
None ⇒ *Mapping.update* (*x*) {*z*} *m* |
Some zs ⇒ *Mapping.update* (*x*) (*Set.insert z zs*) *m*)
xs
Mapping.empty) |
None ⇒ *Code.abort* (*STR ''set-as-map RBT-set*: *ccompare* = *None''*)
(λ-. *set-as-map* (*DList-set xs*)))
(**is** *?C2*)
⟨*proof*⟩

end

3 Underlying FSM Representation

This theory contains the underlying datatype for (possibly not well-formed) finite state machines.

theory *FSM-Impl*
imports *Util Datatype-Order-Generator.Order-Generator HOL−Library.FSet*
begin

A finite state machine (FSM) is represented using its classical definition:

datatype ('*state*, '*input*, '*output*) *fsm-impl* = *FSMI* (*initial* : '*state*)
(*states* : '*state set*)
(*inputs* : '*input set*)
(*outputs* : '*output set*)
(*transitions* : ('*state* × '*input* × '*output* × '*state*) *set*)

3.1 Types for Transitions and Paths

type-synonym ('*a*,'*b*,'*c*) *transition* = ('*a* × '*b* × '*c* × '*a*)
type-synonym ('*a*,'*b*,'*c*) *path* = ('*a*,'*b*,'*c*) *transition list*

abbreviation *t-source* (*a* :: ('*a*,'*b*,'*c*) *transition*) ≡ *fst a*
abbreviation *t-input* (*a* :: ('*a*,'*b*,'*c*) *transition*) ≡ *fst* (*snd a*)
abbreviation *t-output* (*a* :: ('*a*,'*b*,'*c*) *transition*) ≡ *fst* (*snd* (*snd a*))
abbreviation *t-target* (*a* :: ('*a*,'*b*,'*c*) *transition*) ≡ *snd* (*snd* (*snd a*))

3.2 Basic Algorithms on FSM

3.2.1 Reading FSMs from Lists

fun *fsm-impl-from-list* :: $'a \Rightarrow$
$('a,'b,'c)$ *transition list* $\Rightarrow$
$('a, 'b, 'c)$ *fsm-impl*
where
fsm-impl-from-list q [] = *FSMI q* {*q*} {} {} {} |
fsm-impl-from-list q (*t*#*ts*) =
(*let ts′* = *set* (*t*#*ts*)
in FSMI (*t-source t*)
((*image t-source ts′*) ∪ (*image t-target ts′*))
(*image t-input ts′*)
(*image t-output ts′*)
(*ts′*))

fun *fsm-impl-from-list′* :: $'a \Rightarrow ('a,'b,'c)$ *transition list* $\Rightarrow ('a, 'b, 'c)$ *fsm-impl*
where
fsm-impl-from-list′ q [] = *FSMI q* {*q*} {} {} {} |
fsm-impl-from-list′ q (*t*#*ts*) = (*let tsr* = (*remdups* (*t*#*ts*))
in FSMI (*t-source t*)
(*set* (*remdups* ((*map t-source tsr*) @ (*map t-target tsr*))))
(*set* (*remdups* (*map t-input tsr*)))
(*set* (*remdups* (*map t-output tsr*)))
(*set tsr*))

lemma *fsm-impl-from-list-code*[*code*] :
fsm-impl-from-list q ts = *fsm-impl-from-list′ q ts*
⟨*proof*⟩

3.2.2 Changing the initial State

fun *from-FSMI* :: $('a,'b,'c)$ *fsm-impl* $\Rightarrow 'a \Rightarrow ('a,'b,'c)$ *fsm-impl* **where**
from-FSMI M q = (*if q* ∈ *states M then FSMI q* (*states M*) (*inputs M*) (*outputs M*) (*transitions M*) *else M*)

3.2.3 Product Construction

fun *product* :: $('a,'b,'c)$ *fsm-impl* $\Rightarrow ('d,'b,'c)$ *fsm-impl* $\Rightarrow ('a \times 'd,'b,'c)$ *fsm-impl*
where
product A B = *FSMI* ((*initial A*, *initial B*))
((*states A*) × (*states B*))
(*inputs A* ∪ *inputs B*)
(*outputs A* ∪ *outputs B*)
{((*qA*,*qB*),*x*,*y*,(*qA′*,*qB′*)) | *qA qB x y qA′ qB′* . (*qA*,*x*,*y*,*qA′*) ∈ *transitions A* ∧ (*qB*,*x*,*y*,*qB′*) ∈ *transitions B*}

lemma *product-code-naive*[*code*] :

product A B = *FSMI* ((*initial A*, *initial B*))
((*states A*) × (*states B*))
(*inputs A* ∪ *inputs B*)
(*outputs A* ∪ *outputs B*)
(*image* (λ((*qA*,*x*,*y*,*qA′*), (*qB*,*x′*,*y′*,*qB′*)) . ((*qA*,*qB*),*x*,*y*,(*qA′*,*qB′*))) (*Set.filter* (λ((*qA*,*x*,*y*,*qA′*), (*qB*,*x′*,*y′*,*qB′*)) . *x* = *x′* ∧ *y* = *y′*) (⋃(*image* (λ *tA* . *image* (λ *tB* . (*tA*,*tB*)) (*transitions B*)) (*transitions A*)))))
(**is** *?P1* = *?P2*)
⟨*proof*⟩

3.2.4 Filtering Transitions

fun *filter-transitions* :: (′*a*,′*b*,′*c*) *fsm-impl* ⇒ ((′*a*,′*b*,′*c*) *transition* ⇒ *bool*) ⇒ (′*a*,′*b*,′*c*) *fsm-impl* **where**
filter-transitions M P = *FSMI* (*initial M*)
(*states M*)
(*inputs M*)
(*outputs M*)
(*Set.filter P* (*transitions M*))

3.2.5 Filtering States

fun *filter-states* :: (′*a*,′*b*,′*c*) *fsm-impl* ⇒ (′*a* ⇒ *bool*) ⇒ (′*a*,′*b*,′*c*) *fsm-impl* **where**
filter-states M P = (*if P* (*initial M*) *then FSMI* (*initial M*)
(*Set.filter P* (*states M*))
(*inputs M*)
(*outputs M*)
(*Set.filter* (λ *t* . *P* (*t-source t*) ∧ *P* (*t-target t*)) (*transitions M*))
else M)

3.2.6 Initial Singleton FSMI (For Trivial Preamble)

fun *initial-singleton* :: (′*a*,′*b*,′*c*) *fsm-impl* ⇒ (′*a*,′*b*,′*c*) *fsm-impl* **where**
initial-singleton M = *FSMI* (*initial M*)
{*initial M*}
(*inputs M*)
(*outputs M*)
{}

3.2.7 Canonical Separator

abbreviation *shift-Inl t* ≡ (*Inl* (*t-source t*),*t-input t*, *t-output t*, *Inl* (*t-target t*))

definition *shifted-transitions* :: ((′*a* × ′*a*) × ′*b* × ′*c* × (′*a* × ′*a*)) *set* ⇒ (((′*a* × ′*a*) + ′*d*) × ′*b* × ′*c* × ((′*a* × ′*a*) + ′*d*)) *set* **where**
shifted-transitions ts = *image shift-Inl ts*

definition *distinguishing-transitions* :: ((′*a* × ′*b*) ⇒ ′*c set*) ⇒ ′*a* ⇒ ′*a* ⇒ (′*a* × ′*a*) *set* ⇒ ′*b set* ⇒ (((′*a* × ′*a*) + ′*a*) × ′*b* × ′*c* × ((′*a* × ′*a*) + ′*a*)) *set* **where**

distinguishing-transitions f q1 q2 stateSet inputSet = $\bigcup$ (*Set.image* (λ((*q1′*,*q2′*),*x*)
.
(*image* (λ*y* . (*Inl* (*q1′*,*q2′*),*x*,*y*,*Inr* *q1*)) (*f* (*q1′*,*x*) − *f* (*q2′*,*x*)))
∪ (*image* (λ*y* . (*Inl* (*q1′*,*q2′*),*x*,*y*,*Inr* *q2*)) (*f* (*q2′*,*x*) − *f* (*q1′*,*x*))))
(*stateSet* × *inputSet*))

fun *canonical-separator′* :: (*′a*,*′b*,*′c*) *fsm-impl* ⇒ ((*′a* × *′a*),*′b*,*′c*) *fsm-impl* ⇒ *′a* ⇒ *′a* ⇒ ((*′a* × *′a*) + *′a*,*′b*,*′c*) *fsm-impl* **where**
canonical-separator′ M P q1 q2 = (*if initial P* = (*q1*,*q2*)
then
(*let f′* = *set-as-map* (*image* (λ(*q*,*x*,*y*,*q′*) . ((*q*,*x*),*y*)) (*transitions M*));
f = (λ*qx* . (*case f′ qx of Some yqs* ⇒ *yqs* | *None* ⇒ {}));
shifted-transitions′ = *shifted-transitions* (*transitions P*);
distinguishing-transitions-lr = *distinguishing-transitions f q1 q2* (*states P*) (*inputs P*);
ts = *shifted-transitions′* ∪ *distinguishing-transitions-lr*
in

FSMI (*Inl* (*q1*,*q2*))
((*image Inl* (*states P*)) ∪ {*Inr q1*, *Inr q2*})
(*inputs M* ∪ *inputs P*)
(*outputs M* ∪ *outputs P*)
(*ts*))
else FSMI (*Inl* (*q1*,*q2*)) {*Inl* (*q1*,*q2*)} {} {} {})

lemma *h-out-impl-helper*: (λ (*q*,*x*) . {*y* . ∃ *q′* . (*q*,*x*,*y*,*q′*) ∈ *A*}) = (λ*qx* . (*case* (*set-as-map* (*image* (λ(*q*,*x*,*y*,*q′*) . ((*q*,*x*),*y*)) *A*)) *qx of Some yqs* ⇒ *yqs* | *None* ⇒ {}))
⟨*proof*⟩

lemma *canonical-separator′-simps* :
initial (*canonical-separator′ M P q1 q2*) = *Inl* (*q1*,*q2*)
states (*canonical-separator′ M P q1 q2*) = (*if initial P* = (*q1*,*q2*) *then* (*image Inl* (*states P*)) ∪ {*Inr q1*, *Inr q2*} *else* {*Inl* (*q1*,*q2*)})
inputs (*canonical-separator′ M P q1 q2*) = (*if initial P* = (*q1*,*q2*) *then inputs M* ∪ *inputs P else* {})
outputs (*canonical-separator′ M P q1 q2*) = (*if initial P* = (*q1*,*q2*) *then outputs M* ∪ *outputs P else* {})
transitions (*canonical-separator′ M P q1 q2*) = (*if initial P* = (*q1*,*q2*) *then shifted-transitions* (*transitions P*) ∪ *distinguishing-transitions* (λ (*q*,*x*) . {*y* . ∃ *q′* . (*q*,*x*,*y*,*q′*) ∈ *transitions M*}) *q1 q2* (*states P*) (*inputs P*) *else* {})
⟨*proof*⟩

3.2.8 Generalised Canonical Separator

A variation on the state separator that uses states *L* and *R* instead of *Inr q1* and *Inr q2* to indicate targets of transitions in the canonical separator that are available only for the left or right component of a state pair

Note: this definition of a canonical separator might serve as a way to avoid recalculation of state separators for different pairs of states, but is currently not fully implemented

```
datatype LR = Left | Right
derive linorder LR

definition distinguishing-transitions-LR :: (('a × 'b) ⇒ 'c set) ⇒ ('a × 'a) set ⇒
'b set ⇒ ((('a × 'a) + LR) × 'b × 'c × (('a × 'a) + LR)) set where
  distinguishing-transitions-LR f stateSet inputSet = ⋃ (Set.image (λ((q1',q2'),x)
.
                                                   (image (λy . (Inl (q1',q2'),x,y,Inr
Left)) (f (q1',x) − f (q2',x)))
                                                 ∪ (image (λy . (Inl (q1',q2'),x,y,Inr
Right)) (f (q2',x) − f (q1',x))))
                                                   (stateSet × inputSet))


fun canonical-separator-complete' :: ('a,'b,'c) fsm-impl ⇒ (('a × 'a) + LR,'b,'c)
fsm-impl where
  canonical-separator-complete' M =
    (let P = product M M;
         f'  = set-as-map (image (λ(q,x,y,q') . ((q,x),y)) (transitions M));
         f   = (λqx . (case f' qx of Some yqs ⇒ yqs | None ⇒ {}));
         shifted-transitions' = shifted-transitions (transitions P);
           distinguishing-transitions-lr = distinguishing-transitions-LR f (states P)
(inputs P);
         ts = shifted-transitions' ∪ distinguishing-transitions-lr
     in
      FSMI (Inl (initial P))
          ((image Inl (states P)) ∪ {Inr Left, Inr Right})
          (inputs M ∪ inputs P)
          (outputs M ∪ outputs P)
          ts )
```

3.2.9 Adding Transitions

```
fun add-transitions :: ('a,'b,'c) fsm-impl ⇒ ('a,'b,'c) transition set ⇒ ('a,'b,'c)
fsm-impl where
  add-transitions M ts = (if (∀ t ∈ ts . t-source t ∈ states M ∧ t-input t ∈ inputs
M ∧ t-output t ∈ outputs M ∧ t-target t ∈ states M)
    then FSMI (initial M)
             (states M)
             (inputs M)
```

(*outputs M*)
((*transitions M*) ∪ *ts*)
else M)

3.2.10 Creating an FSMI without transitions

fun *create-unconnected-FSMI* :: *'a* ⇒ *'a set* ⇒ *'b set* ⇒ *'c set* ⇒ (*'a*,*'b*,*'c*) *fsm-impl* **where**
create-unconnected-FSMI q ns ins outs = (*if* (*finite ns* ∧ *finite ins* ∧ *finite outs*)
then FSMI q (*insert q ns*) *ins outs* {}
else FSMI q {*q*} {} {} {})

fun *create-unconnected-fsm-from-lists* :: *'a* ⇒ *'a list* ⇒ *'b list* ⇒ *'c list* ⇒ (*'a*,*'b*,*'c*) *fsm-impl* **where**
create-unconnected-fsm-from-lists q ns ins outs = *FSMI q* (*insert q* (*set ns*)) (*set ins*) (*set outs*) {}

fun *create-unconnected-fsm-from-fsets* :: *'a* ⇒ *'a fset* ⇒ *'b fset* ⇒ *'c fset* ⇒ (*'a*,*'b*,*'c*) *fsm-impl* **where**
create-unconnected-fsm-from-fsets q ns ins outs = *FSMI q* (*insert q* (*fset ns*)) (*fset ins*) (*fset outs*) {}

fun *create-fsm-from-sets* :: *'a* ⇒ *'a set* ⇒ *'b set* ⇒ *'c set* ⇒ (*'a*,*'b*,*'c*) *transition set* ⇒ (*'a*,*'b*,*'c*) *fsm-impl* **where**
create-fsm-from-sets q qs ins outs ts = (*if q* ∈ *qs* ∧ *finite qs* ∧ *finite ins* ∧ *finite outs*
then add-transitions (*FSMI q qs ins outs* {}) *ts*
else FSMI q {*q*} {} {} {})

3.3 Transition Function h

Function h represents the classical view of the transition relation of an FSM M as a function: given a state q and an input x, $(h\ M)\ (q,x)$ returns all possibly reactions (y,q') of M in state q to x, where y is the produced output and q' the target state of the reaction transition.

fun *h* :: (*'state*, *'input*, *'output*) *fsm-impl* ⇒ (*'state* × *'input*) ⇒ (*'output* × *'state*) *set* **where**
h M (*q*,*x*) = { (*y*,*q'*) . (*q*,*x*,*y*,*q'*) ∈ *transitions M* }

fun *h-obs* :: (*'a*,*'b*,*'c*) *fsm-impl* ⇒ *'a* ⇒ *'b* ⇒ *'c* ⇒ *'a option* **where**
h-obs M q x y = (*let*
tgts = *snd* ' *Set.filter* (λ (*y'*,*q'*) . *y'* = *y*) (*h M* (*q*,*x*))
in if card tgts = *1*
then Some (*the-elem tgts*)
else None)

lemma *h-code*[*code*] :
h M (*q*,*x*) = (*let m* = *set-as-map* (*image* (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*,*q'*)) (*transitions M*))

in (*case m* (*q*,*x*) *of Some yqs* ⇒ *yqs* | *None* ⇒ {}))
⟨*proof*⟩

3.4 Extending FSMs by single elements

fun *add-transition* :: ('*a*,'*b*,'*c*) *fsm-impl* ⇒
('*a*,'*b*,'*c*) *transition* ⇒
('*a*,'*b*,'*c*) *fsm-impl*
where
add-transition M t =
(*if t-source t* ∈ *states M* ∧ *t-input t* ∈ *inputs M* ∧
t-output t ∈ *outputs M* ∧ *t-target t* ∈ *states M*
then FSMI (*initial M*)
(*states M*)
(*inputs M*)
(*outputs M*)
(*insert t* (*transitions M*))
else M)

fun *add-state* :: ('*a*,'*b*,'*c*) *fsm-impl* ⇒ '*a* ⇒ ('*a*,'*b*,'*c*) *fsm-impl* **where**
add-state M q = *FSMI* (*initial M*) (*insert q* (*states M*)) (*inputs M*) (*outputs M*) (*transitions M*)

fun *add-input* :: ('*a*,'*b*,'*c*) *fsm-impl* ⇒ '*b* ⇒ ('*a*,'*b*,'*c*) *fsm-impl* **where**
add-input M x = *FSMI* (*initial M*) (*states M*) (*insert x* (*inputs M*)) (*outputs M*) (*transitions M*)

fun *add-output* :: ('*a*,'*b*,'*c*) *fsm-impl* ⇒ '*c* ⇒ ('*a*,'*b*,'*c*) *fsm-impl* **where**
add-output M y = *FSMI* (*initial M*) (*states M*) (*inputs M*) (*insert y* (*outputs M*)) (*transitions M*)

fun *add-transition-with-components* :: ('*a*,'*b*,'*c*) *fsm-impl* ⇒ ('*a*,'*b*,'*c*) *transition* ⇒ ('*a*,'*b*,'*c*) *fsm-impl* **where**
add-transition-with-components M t = *add-transition* (*add-state* (*add-state* (*add-input* (*add-output M* (*t-output t*)) (*t-input t*)) (*t-source t*)) (*t-target t*)) *t*

3.5 Renaming elements

fun *rename-states* :: ('*a*,'*b*,'*c*) *fsm-impl* ⇒ ('*a* ⇒ '*d*) ⇒ ('*d*,'*b*,'*c*) *fsm-impl* **where**
rename-states M f = *FSMI* (*f* (*initial M*))
(*f* ' *states M*)
(*inputs M*)
(*outputs M*)
((λ*t* . (*f* (*t-source t*), *t-input t*, *t-output t*, *f* (*t-target t*))) ' *transitions M*)

end

4 Finite State Machines

This theory defines well-formed finite state machines and introduces various closely related notions, as well as a selection of basic properties and definitions.

theory *FSM*
imports *FSM-Impl HOL−Library.Quotient-Type HOL−Library.Product-Lexorder*
begin

4.1 Well-formed Finite State Machines

A value of type *fsm-impl* constitutes a well-formed FSM if its contained sets are finite and the initial state and the components of each transition are contained in their respective sets.

abbreviation(*input*) *well-formed-fsm* (M :: ($'state$, $'input$, $'output$) *fsm-impl*)
≡ (*initial M* ∈ *states M*
∧ *finite* (*states M*)
∧ *finite* (*inputs M*)
∧ *finite* (*outputs M*)
∧ *finite* (*transitions M*)
∧ (∀ *t* ∈ *transitions M* . *t-source t* ∈ *states M* ∧
t-input t ∈ *inputs M* ∧
t-target t ∈ *states M* ∧
t-output t ∈ *outputs M*))

typedef ($'state$, $'input$, $'output$) *fsm* =
{ *M* :: ($'state$, $'input$, $'output$) *fsm-impl* . *well-formed-fsm M*}
morphisms *fsm-impl-of-fsm Abs-fsm*
⟨*proof*⟩

setup-lifting *type-definition-fsm*

lift-definition *initial* :: ($'state$, $'input$, $'output$) *fsm* ⇒ $'state$ **is** *FSM-Impl.initial*
⟨*proof*⟩
lift-definition *states* :: ($'state$, $'input$, $'output$) *fsm* ⇒ $'state$ *set* **is** *FSM-Impl.states*
⟨*proof*⟩
lift-definition *inputs* :: ($'state$, $'input$, $'output$) *fsm* ⇒ $'input$ *set* **is** *FSM-Impl.inputs*
⟨*proof*⟩
lift-definition *outputs* :: ($'state$, $'input$, $'output$) *fsm* ⇒ $'output$ *set* **is** *FSM-Impl.outputs*
⟨*proof*⟩
lift-definition *transitions* ::
($'state$, $'input$, $'output$) *fsm* ⇒ ($'state$ × $'input$ × $'output$ × $'state$) *set*
is *FSM-Impl.transitions* ⟨*proof*⟩

lift-definition *fsm-from-list* :: $'a$ ⇒ ($'a$,$'b$,$'c$) *transition list* ⇒ ($'a$, $'b$, $'c$) *fsm*
is *FSM-Impl.fsm-impl-from-list*
⟨*proof*⟩

lemma *fsm-initial*[*intro*]: *initial M* ∈ *states M*
⟨*proof*⟩
lemma *fsm-states-finite*: *finite* (*states M*)
⟨*proof*⟩
lemma *fsm-inputs-finite*: *finite* (*inputs M*)
⟨*proof*⟩
lemma *fsm-outputs-finite*: *finite* (*outputs M*)
⟨*proof*⟩
lemma *fsm-transitions-finite*: *finite* (*transitions M*)
⟨*proof*⟩
lemma *fsm-transition-source*[*intro*]: ⋀ *t* . *t* ∈ (*transitions M*) ⟹ *t-source t* ∈ *states M*
⟨*proof*⟩
lemma *fsm-transition-target*[*intro*]: ⋀ *t* . *t* ∈ (*transitions M*) ⟹ *t-target t* ∈ *states M*
⟨*proof*⟩
lemma *fsm-transition-input*[*intro*]: ⋀ *t* . *t* ∈ (*transitions M*) ⟹ *t-input t* ∈ *inputs M*
⟨*proof*⟩
lemma *fsm-transition-output*[*intro*]: ⋀ *t* . *t* ∈ (*transitions M*) ⟹ *t-output t* ∈ *outputs M*
⟨*proof*⟩

instantiation *fsm* :: (*type*,*type*,*type*) *equal*
begin
definition *equal-fsm* :: (*'a*, *'b*, *'c*) *fsm* ⇒ (*'a*, *'b*, *'c*) *fsm* ⇒ *bool* **where**
equal-fsm x y = (*initial x* = *initial y* ∧ *states x* = *states y* ∧ *inputs x* = *inputs y* ∧ *outputs x* = *outputs y* ∧ *transitions x* = *transitions y*)

instance
⟨*proof*⟩
end

4.1.1 Example FSMs

definition *m-ex-H* :: (*integer*,*integer*,*integer*) *fsm* **where**
m-ex-H = *fsm-from-list 1* [(*1*,*0*,*0*,*2*),
(*1*,*0*,*1*,*4*),
(*1*,*1*,*1*,*4*),
(*2*,*0*,*0*,*2*),
(*2*,*1*,*1*,*4*),
(*3*,*0*,*1*,*4*),
(*3*,*1*,*0*,*1*),
(*3*,*1*,*1*,*3*),
(*4*,*0*,*0*,*3*),

(4,1,0,1)]

definition *m-ex-9* :: (*integer*,*integer*,*integer*) *fsm* **where**
m-ex-9 = *fsm-from-list 0* [(0,0,2,2),
(0,0,3,2),
(0,1,0,3),
(0,1,1,3),
(1,0,3,2),
(1,1,1,3),
(2,0,2,2),
(2,1,3,3),
(3,0,2,2),
(3,1,0,2),
(3,1,1,1)]

definition *m-ex-DR* :: (*integer*,*integer*,*integer*) *fsm* **where**
m-ex-DR = *fsm-from-list 0* [(0,0,0,100),
(100,0,0,101),
(100,0,1,101),
(101,0,0,102),
(101,0,1,102),
(102,0,0,103),
(102,0,1,103),
(103,0,0,104),
(103,0,1,104),
(104,0,0,100),
(104,0,1,100),
(104,1,0,400),
(0,0,2,200),
(200,0,2,201),
(201,0,2,202),
(202,0,2,203),
(203,0,2,200),
(203,1,0,400),
(0,1,0,300),
(100,1,0,300),
(101,1,0,300),
(102,1,0,300),
(103,1,0,300),
(200,1,0,300),
(201,1,0,300),
(202,1,0,300),
(300,0,0,300),
(300,1,0,300),
(400,0,0,300),
(400,1,0,300)]

4.2 Transition Function h and related functions

lift-definition *h* :: ('*state*, '*input*, '*output*) *fsm* ⇒ ('*state* × '*input*) ⇒ ('*output* × '*state*) *set*
  is *FSM-Impl.h* ⟨*proof*⟩

lemma *h-simps*[*simp*]: *FSM.h M* (*q*,*x*) = { (*y*,*q*′) . (*q*,*x*,*y*,*q*′) ∈ *transitions M* }
  ⟨*proof*⟩

lift-definition *h-obs* :: ('*state*, '*input*, '*output*) *fsm* ⇒ '*state* ⇒ '*input* ⇒ '*output* ⇒ '*state option*
  is *FSM-Impl.h-obs* ⟨*proof*⟩

lemma *h-obs-simps*[*simp*]: *FSM.h-obs M q x y* = (*let*
    tgts = *snd* ' *Set.filter* (λ (*y*′,*q*′) . *y*′ = *y*) (*h M* (*q*,*x*))
  in if card tgts = *1*
    then Some (*the-elem tgts*)
    else None)
  ⟨*proof*⟩

fun *defined-inputs*′ :: (('*a* ×'*b*) ⇒ ('*c*×'*a*) *set*) ⇒ '*b set* ⇒ '*a* ⇒ '*b set* **where**
  defined-inputs′ *hM iM q* = {*x* ∈ *iM* . *hM* (*q*,*x*) ≠ {}}

fun *defined-inputs* :: ('*a*,'*b*,'*c*) *fsm* ⇒ '*a* ⇒ '*b set* **where**
  defined-inputs M q = *defined-inputs*′ (*h M*) (*inputs M*) *q*

lemma *defined-inputs-set* : *defined-inputs M q* = {*x* ∈ *inputs M* . *h M* (*q*,*x*) ≠ {}
}
  ⟨*proof*⟩

fun *transitions-from*′ :: (('*a* ×'*b*) ⇒ ('*c*×'*a*) *set*) ⇒ '*b set* ⇒ '*a* ⇒ ('*a*,'*b*,'*c*) *transition set* **where**
  transitions-from′ *hM iM q* = ⋃(*image* (λ*x* . *image* (λ(*y*,*q*′) . (*q*,*x*,*y*,*q*′)) (*hM* (*q*,*x*))) *iM*)

fun *transitions-from* :: ('*a*,'*b*,'*c*) *fsm* ⇒ '*a* ⇒ ('*a*,'*b*,'*c*) *transition set* **where**
  transitions-from M q = *transitions-from*′ (*h M*) (*inputs M*) *q*

lemma *transitions-from-set* :
  assumes *q* ∈ *states M*
  shows *transitions-from M q* = {*t* ∈ *transitions M* . *t-source t* = *q*}
⟨*proof*⟩

fun *h-from* :: ('*state*, '*input*, '*output*) *fsm* ⇒ '*state* ⇒ ('*input* × '*output* × '*state*) *set* **where**
  h-from M q = { (*x*,*y*,*q*′) . (*q*,*x*,*y*,*q*′) ∈ *transitions M* }

lemma *h-from*[*code*] : *h-from M q* = (*let m* = *set-as-map* (*transitions M*)
in (*case m q of Some yqs* ⇒ *yqs* | *None* ⇒ {}))
⟨*proof*⟩

fun *h-out* :: ('*a*,'*b*,'*c*) *fsm* ⇒ ('*a* × '*b*) ⇒ '*c set* **where**
h-out M (*q*,*x*) = {*y* . ∃ *q'* . (*q*,*x*,*y*,*q'*) ∈ *transitions M*}

lemma *h-out-code*[*code*]:
h-out M = (λ*qx* . (*case* (*set-as-map* (*image* (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*)) (*transitions M*))) *qx of*
Some yqs ⇒ *yqs* |
None ⇒ {}))
⟨*proof*⟩

lemma *h-out-alt-def* :
h-out M (*q*,*x*) = {*t-output t* | *t* . *t* ∈ *transitions M* ∧ *t-source t* = *q* ∧ *t-input t* = *x*}
⟨*proof*⟩

4.3 Size

instantiation *fsm* :: (*type*,*type*,*type*) *size*
begin

definition *size* **where** [*simp*, *code*]: *size* (*m*::('*a*, '*b*, '*c*) *fsm*) = *card* (*states m*)

instance ⟨*proof*⟩
end

lemma *fsm-size-Suc* :
size M > *0*
⟨*proof*⟩

4.4 Paths

inductive *path* :: ('*state*, '*input*, '*output*) *fsm* ⇒ '*state* ⇒ ('*state*, '*input*, '*output*) *path* ⇒ *bool*
where
nil[*intro*!] : *q* ∈ *states M* ⟹ *path M q* [] |
cons[*intro*!] : *t* ∈ *transitions M* ⟹ *path M* (*t-target t*) *ts* ⟹ *path M* (*t-source t*) (*t*#*ts*)

inductive-cases *path-nil-elim*[*elim*!]: *path M q* []
inductive-cases *path-cons-elim*[*elim*!]: *path M q* (*t*#*ts*)

fun *visited-states* :: '*state* ⇒ ('*state*, '*input*, '*output*) *path* ⇒ '*state list* **where**
visited-states q p = (*q* # *map t-target p*)

fun *target* :: '*state* ⇒ ('*state*, '*input*, '*output*) *path* ⇒ '*state* **where**

target q p = *last* (*visited-states q p*)

lemma *target-nil* [*simp*] : *target q* [] = *q* ⟨*proof*⟩
lemma *target-snoc* [*simp*] : *target q* (*p*@[*t*]) = *t-target t* ⟨*proof*⟩

lemma *path-begin-state* :
assumes *path M q p*
shows $q \in$ *states M*
⟨*proof*⟩

lemma *path-append*[*intro*!] :
assumes *path M q p1*
and *path M* (*target q p1*) *p2*
shows *path M q* (*p1*@*p2*)
⟨*proof*⟩

lemma *path-target-is-state* :
assumes *path M q p*
shows *target q p* $\in$ *states M*
⟨*proof*⟩

lemma *path-suffix* :
assumes *path M q* (*p1*@*p2*)
shows *path M* (*target q p1*) *p2*
⟨*proof*⟩

lemma *path-prefix* :
assumes *path M q* (*p1*@*p2*)
shows *path M q p1*
⟨*proof*⟩

lemma *path-append-elim*[*elim*!] :
assumes *path M q* (*p1*@*p2*)
obtains *path M q p1*
and *path M* (*target q p1*) *p2*
⟨*proof*⟩

lemma *path-append-target*:
target q (*p1*@*p2*) = *target* (*target q p1*) *p2*
⟨*proof*⟩

lemma *path-append-target-hd* :
assumes *length p* > *0*
shows *target q p* = *target* (*t-target* (*hd p*)) (*tl p*)
⟨*proof*⟩

lemma *path-transitions* :
assumes *path M q p*

shows *set p* ⊆ *transitions M*
⟨*proof*⟩

lemma *path-append-transition*[*intro*!] :
assumes *path M q p*
and $t \in$ *transitions M*
and *t-source t* = *target q p*
shows *path M q* (*p*@[*t*])
⟨*proof*⟩

lemma *path-append-transition-elim*[*elim*!] :
assumes *path M q* (*p*@[*t*])
shows *path M q p*
and $t \in$ *transitions M*
and *t-source t* = *target q p*
⟨*proof*⟩

lemma *path-prepend-t* : *path M q′ p* ⟹ $(q,x,y,q') \in$ *transitions M* ⟹ *path M q* $((q,x,y,q')\#p)$
⟨*proof*⟩

lemma *path-target-append* : *target q1 p1* = *q2* ⟹ *target q2 p2* = *q3* ⟹ *target q1* (*p1*@*p2*) = *q3*
⟨*proof*⟩

lemma *single-transition-path* : $t \in$ *transitions M* ⟹ *path M* (*t-source t*) [*t*] ⟨*proof*⟩

lemma *path-source-target-index* :
assumes *Suc i* < *length p*
and *path M q p*
shows *t-target* (*p* ! *i*) = *t-source* (*p* ! (*Suc i*))
⟨*proof*⟩

lemma *paths-finite* : *finite* { *p* . *path M q p* ∧ *length p* ≤ *k* }
⟨*proof*⟩

lemma *visited-states-prefix* :
assumes *q′* ∈ *set* (*visited-states q p*)
shows ∃ *p1 p2* . *p* = *p1*@*p2* ∧ *target q p1* = *q′*
⟨*proof*⟩

lemma *visited-states-are-states* :
assumes *path M q1 p*
shows *set* (*visited-states q1 p*) ⊆ *states M*
⟨*proof*⟩

lemma *transition-subset-path* :
assumes *transitions A* ⊆ *transitions B*
and *path A q p*

and $q \in states\ B$
shows $path\ B\ q\ p$
$\langle proof \rangle$

4.4.1 Paths of fixed length

fun $paths\text{-}of\text{-}length'$ :: $('a,'b,'c)\ path \Rightarrow 'a \Rightarrow (('a \times 'b) \Rightarrow ('c \times 'a)\ set) \Rightarrow 'b\ set \Rightarrow nat \Rightarrow ('a,'b,'c)\ path\ set$
where
$paths\text{-}of\text{-}length'\ prev\ q\ hM\ iM\ 0 = \{prev\}$ |
$paths\text{-}of\text{-}length'\ prev\ q\ hM\ iM\ (Suc\ k) =$
$(let\ hF = transitions\text{-}from'\ hM\ iM\ q$
$in \bigcup\ (image\ (\lambda\ t\ .\ paths\text{-}of\text{-}length'\ (prev@[t])\ (t\text{-}target\ t)\ hM\ iM\ k)\ hF))$

fun $paths\text{-}of\text{-}length$ :: $('a,'b,'c)\ fsm \Rightarrow 'a \Rightarrow nat \Rightarrow ('a,'b,'c)\ path\ set$ **where**
$paths\text{-}of\text{-}length\ M\ q\ k = paths\text{-}of\text{-}length'\ []\ q\ (h\ M)\ (inputs\ M)\ k$

4.4.2 Paths up to fixed length

fun $paths\text{-}up\text{-}to\text{-}length'$ :: $('a,'b,'c)\ path \Rightarrow 'a \Rightarrow (('a \times 'b) \Rightarrow (('c \times 'a)\ set)) \Rightarrow 'b\ set \Rightarrow nat \Rightarrow ('a,'b,'c)\ path\ set$
where
$paths\text{-}up\text{-}to\text{-}length'\ prev\ q\ hM\ iM\ 0 = \{prev\}$ |
$paths\text{-}up\text{-}to\text{-}length'\ prev\ q\ hM\ iM\ (Suc\ k) =$
$(let\ hF = transitions\text{-}from'\ hM\ iM\ q$
$in\ insert\ prev\ (\bigcup\ (image\ (\lambda\ t\ .\ paths\text{-}up\text{-}to\text{-}length'\ (prev@[t])\ (t\text{-}target\ t)\ hM\ iM\ k)\ hF)))$

fun $paths\text{-}up\text{-}to\text{-}length$ :: $('a,'b,'c)\ fsm \Rightarrow 'a \Rightarrow nat \Rightarrow ('a,'b,'c)\ path\ set$ **where**
$paths\text{-}up\text{-}to\text{-}length\ M\ q\ k = paths\text{-}up\text{-}to\text{-}length'\ []\ q\ (h\ M)\ (inputs\ M)\ k$

lemma $paths\text{-}up\text{-}to\text{-}length'\text{-}set$:
assumes $q \in states\ M$
and $path\ M\ q\ prev$
shows $paths\text{-}up\text{-}to\text{-}length'\ prev\ (target\ q\ prev)\ (h\ M)\ (inputs\ M)\ k$
$= \{(prev@p) \mid p\ .\ path\ M\ (target\ q\ prev)\ p \wedge length\ p \leq k\}$
$\langle proof \rangle$

lemma $paths\text{-}up\text{-}to\text{-}length\text{-}set$:
assumes $q \in states\ M$
shows $paths\text{-}up\text{-}to\text{-}length\ M\ q\ k = \{p\ .\ path\ M\ q\ p \wedge length\ p \leq k\}$
$\langle proof \rangle$

4.4.3 Calculating Acyclic Paths

fun $acyclic\text{-}paths\text{-}up\text{-}to\text{-}length'$ :: $('a,'b,'c)\ path \Rightarrow 'a \Rightarrow ('a \Rightarrow (('b \times 'c \times 'a)\ set)) \Rightarrow 'a\ set \Rightarrow nat \Rightarrow ('a,'b,'c)\ path\ set$
where

acyclic-paths-up-to-length′ prev q hF visitedStates 0 = {*prev*} |
acyclic-paths-up-to-length′ prev q hF visitedStates (*Suc k*) =
(*let tF* = *Set.filter* (λ (*x*,*y*,*q′*) . *q′* ∉ *visitedStates*) (*hF q*)
in (*insert prev* (⋃ (*image* (λ (*x*,*y*,*q′*) . *acyclic-paths-up-to-length′* (*prev*@[(*q*,*x*,*y*,*q′*)]) *q′ hF* (*insert q′ visitedStates*) *k*) *tF*))))

fun *p-source* :: *′a* ⇒ (*′a*,*′b*,*′c*) *path* ⇒ *′a*
where *p-source q p* = *hd* (*visited-states q p*)

lemma *acyclic-paths-up-to-length′-prev* :
p′ ∈ *acyclic-paths-up-to-length′* (*prev*@*prev′*) *q hF visitedStates k* ⟹ ∃ *p′′* . *p′* = *prev*@*p′′*
⟨*proof*⟩

lemma *acyclic-paths-up-to-length′-set* :
assumes *path M* (*p-source q prev*) *prev*
and ⋀ *q′* . *hF q′* = {(*x*,*y*,*q′′*) | *x y q′′* . (*q′*,*x*,*y*,*q′′*) ∈ *transitions M*}
and *distinct* (*visited-states* (*p-source q prev*) *prev*)
and *visitedStates* = *set* (*visited-states* (*p-source q prev*) *prev*)
shows *acyclic-paths-up-to-length′ prev* (*target* (*p-source q prev*) *prev*) *hF visitedStates k*
= { *prev*@*p* | *p* . *path M* (*p-source q prev*) (*prev*@*p*)
∧ *length p* ≤ *k*
∧ *distinct* (*visited-states* (*p-source q prev*) (*prev*@*p*)) }
⟨*proof*⟩

fun *acyclic-paths-up-to-length* :: (*′a*,*′b*,*′c*) *fsm* ⇒ *′a* ⇒ *nat* ⇒ (*′a*,*′b*,*′c*) *path set*
where
acyclic-paths-up-to-length M q k = {*p*. *path M q p* ∧ *length p* ≤ *k* ∧ *distinct* (*visited-states q p*)}

lemma *acyclic-paths-up-to-length-code*[*code*] :
acyclic-paths-up-to-length M q k = (*if q* ∈ *states M*
then acyclic-paths-up-to-length′ [] *q* (*m2f* (*set-as-map* (*transitions M*))) {*q*} *k*
else {})
⟨*proof*⟩

lemma *path-map-target* : *target* (*f4 q*) (*map* (λ *t* . (*f1* (*t-source t*), *f2* (*t-input t*), *f3* (*t-output t*), *f4* (*t-target t*))) *p*) = *f4* (*target q p*)
⟨*proof*⟩

lemma *path-length-sum* :
assumes *path M q p*
shows *length p* = (∑ *q* ∈ *states M* . *length* (*filter* (λ*t*. *t-target t* = *q*) *p*))
⟨*proof*⟩

lemma *path-loop-cut* :
assumes *path M q p*
and *t-target* (*p* ! *i*) = *t-target* (*p* ! *j*)
and *i* < *j*
and *j* < *length p*
shows *path M q* ((*take* (*Suc i*) *p*) @ (*drop* (*Suc j*) *p*))
and *target q* ((*take* (*Suc i*) *p*) @ (*drop* (*Suc j*) *p*)) = *target q p*
and *length* ((*take* (*Suc i*) *p*) @ (*drop* (*Suc j*) *p*)) < *length p*
and *path M* (*target q* (*take* (*Suc i*) *p*)) (*drop* (*Suc i*) (*take* (*Suc j*) *p*))
and *target* (*target q* (*take* (*Suc i*) *p*)) (*drop* (*Suc i*) (*take* (*Suc j*) *p*)) = (*target q* (*take* (*Suc i*) *p*))
⟨*proof*⟩

lemma *path-prefix-take* :
assumes *path M q p*
shows *path M q* (*take i p*)
⟨*proof*⟩

4.5 Acyclic Paths

lemma *cyclic-path-loop* :
assumes *path M q p*
and ¬ *distinct* (*visited-states q p*)
shows ∃ *p1 p2 p3* . *p* = *p1*@*p2*@*p3* ∧ *p2* ≠ [] ∧ *target q p1* = *target q* (*p1*@*p2*)
⟨*proof*⟩

lemma *cyclic-path-pumping* :
assumes *path M* (*initial M*) *p*
and ¬ *distinct* (*visited-states* (*initial M*) *p*)
shows ∃ *p* . *path M* (*initial M*) *p* ∧ *length p* ≥ *n*
⟨*proof*⟩

lemma *cyclic-path-shortening* :
assumes *path M q p*
and ¬ *distinct* (*visited-states q p*)
shows ∃ *p′* . *path M q p′* ∧ *target q p′* = *target q p* ∧ *length p′* < *length p*
⟨*proof*⟩

lemma *acyclic-path-from-cyclic-path* :
assumes *path M q p*
and ¬ *distinct* (*visited-states q p*)
obtains *p′* **where** *path M q p′* **and** *target q p* = *target q p′* **and** *distinct* (*visited-states q p′*)

⟨*proof*⟩

lemma *acyclic-path-length-limit* :
assumes *path M q p*
and *distinct* (*visited-states q p*)
shows *length p* < *size M*
⟨*proof*⟩

4.6 Reachable States

definition *reachable* :: (′*a*,′*b*,′*c*) *fsm* ⇒ ′*a* ⇒ *bool* **where**
reachable M q = (∃ *p* . *path M* (*initial M*) *p* ∧ *target* (*initial M*) *p* = *q*)

definition *reachable-states* :: (′*a*,′*b*,′*c*) *fsm* ⇒ ′*a set* **where**
reachable-states M = {*target* (*initial M*) *p* | *p* . *path M* (*initial M*) *p* }

abbreviation *size-r M* ≡ *card* (*reachable-states M*)

lemma *acyclic-paths-set* :
acyclic-paths-up-to-length M q (*size M* − *1*) = {*p* . *path M q p* ∧ *distinct* (*visited-states q p*)}
⟨*proof*⟩

lemma *reachable-states-code*[*code*] :
reachable-states M = *image* (*target* (*initial M*)) (*acyclic-paths-up-to-length M* (*initial M*) (*size M* − *1*))
⟨*proof*⟩

lemma *reachable-states-intro*[*intro*!] :
assumes *path M* (*initial M*) *p*
shows *target* (*initial M*) *p* ∈ *reachable-states M*
⟨*proof*⟩

lemma *reachable-states-initial* :
initial M ∈ *reachable-states M*
⟨*proof*⟩

lemma *reachable-states-next* :
assumes *q* ∈ *reachable-states M* **and** *t* ∈ *transitions M* **and** *t-source t* = *q*
shows *t-target t* ∈ *reachable-states M*
⟨*proof*⟩

lemma *reachable-states-path* :
assumes $q \in$ *reachable-states M*
and *path M q p*
and $t \in$ *set p*
shows *t-source t* $\in$ *reachable-states M*
⟨*proof*⟩

lemma *reachable-states-initial-or-target* :
assumes $q \in$ *reachable-states M*
shows $q =$ *initial M* $\vee$ ($\exists$ $t \in$ *transitions M* . *t-source t* $\in$ *reachable-states M* $\wedge$ *t-target t* $= q$)
⟨*proof*⟩

lemma *reachable-state-is-state* :
$q \in$ *reachable-states M* $\Longrightarrow$ $q \in$ *states M*
⟨*proof*⟩

lemma *reachable-states-finite* : *finite* (*reachable-states M*)
⟨*proof*⟩

4.7 Language

abbreviation *p-io* (*p* :: (*'state,'input,'output*) *path*) $\equiv$ *map* (λ *t* . (*t-input t*, *t-output t*)) *p*

fun *language-state-for-input* :: (*'state,'input,'output*) *fsm* $\Rightarrow$ *'state* $\Rightarrow$ *'input list* $\Rightarrow$ (*'input* $\times$ *'output*) *list set* **where**
language-state-for-input M q xs = {*p-io p* | *p* . *path M q p* $\wedge$ *map fst* (*p-io p*) = *xs*}

fun LS_{in} :: (*'state,'input,'output*) *fsm* $\Rightarrow$ *'state* $\Rightarrow$ *'input list set* $\Rightarrow$ (*'input* $\times$ *'output*) *list set* **where**
LS_{in} *M q xss* = {*p-io p* | *p* . *path M q p* $\wedge$ *map fst* (*p-io p*) $\in$ *xss*}

abbreviation(*input*) L_{in} *M* $\equiv$ LS_{in} *M* (*initial M*)

lemma *language-state-for-input-inputs* :
assumes *io* $\in$ *language-state-for-input M q xs*
shows *map fst io* = *xs*
⟨*proof*⟩

lemma *language-state-for-inputs-inputs* :
assumes *io* $\in$ LS_{in} *M q xss*
shows *map fst io* $\in$ *xss* ⟨*proof*⟩

fun *LS* :: (*'state,'input,'output*) *fsm* $\Rightarrow$ *'state* $\Rightarrow$ (*'input* $\times$ *'output*) *list set* **where**

LS M q = { *p-io p* | *p* . *path M q p* }

abbreviation *L M* ≡ *LS M* (*initial M*)

lemma *language-state-containment* :
assumes *path M q p*
and *p-io p* = *io*
shows *io* ∈ *LS M q*
⟨*proof*⟩

lemma *language-prefix* :
assumes *io1*@*io2* ∈ *LS M q*
shows *io1* ∈ *LS M q*
⟨*proof*⟩

lemma *language-contains-empty-sequence* : [] ∈ *L M*
⟨*proof*⟩

lemma *language-state-split* :
assumes *io1* @ *io2* ∈ *LS M q1*
obtains *p1 p2* **where** *path M q1 p1*
and *path M* (*target q1 p1*) *p2*
and *p-io p1* = *io1*
and *p-io p2* = *io2*
⟨*proof*⟩

lemma *language-initial-path-append-transition* :
assumes *ios* @ [*io*] ∈ *L M*
obtains *p t* **where** *path M* (*initial M*) (*p*@[*t*]) **and** *p-io* (*p*@[*t*]) = *ios* @ [*io*]
⟨*proof*⟩

lemma *language-path-append-transition* :
assumes *ios* @ [*io*] ∈ *LS M q*
obtains *p t* **where** *path M q* (*p*@[*t*]) **and** *p-io* (*p*@[*t*]) = *ios* @ [*io*]
⟨*proof*⟩

lemma *language-split* :
assumes *io1*@*io2* ∈ *L M*
obtains *p1 p2* **where** *path M* (*initial M*) (*p1*@*p2*) **and** *p-io p1* = *io1* **and** *p-io p2* = *io2*
⟨*proof*⟩

lemma *language-io* :
assumes *io* ∈ *LS M q*

and $(x,y) \in$ *set io*
shows $x \in$ (*inputs M*)
and $y \in$ *outputs M*
⟨*proof*⟩

lemma *path-io-split* :
assumes *path M q p*
and *p-io p* = *io1*@*io2*
shows *path M q* (*take* (*length io1*) *p*)
and *p-io* (*take* (*length io1*) *p*) = *io1*
and *path M* (*target q* (*take* (*length io1*) *p*)) (*drop* (*length io1*) *p*)
and *p-io* (*drop* (*length io1*) *p*) = *io2*
⟨*proof*⟩

lemma *language-intro* :
assumes *path M q p*
shows *p-io p* $\in$ *LS M q*
⟨*proof*⟩

lemma *language-prefix-append* :
assumes *io1* @ (*p-io p*) $\in$ *L M*
shows *io1* @ *p-io* (*take i p*) $\in$ *L M*
⟨*proof*⟩

lemma *language-finite*: *finite* {*io* . *io* $\in$ *L M* $\wedge$ *length io* $\leq$ *k*}
⟨*proof*⟩

lemma *LS-prepend-transition* :
assumes $t \in$ *transitions M*
and *io* $\in$ *LS M* (*t-target t*)
shows (*t-input t*, *t-output t*) # *io* $\in$ *LS M* (*t-source t*)
⟨*proof*⟩

lemma *language-empty-IO* :
assumes *inputs M* = {} $\vee$ *outputs M* = {}
shows *L M* = {[]}
⟨*proof*⟩

lemma *language-equivalence-from-isomorphism-helper* :
assumes *bij-betw f* (*states M1*) (*states M2*)
and *f* (*initial M1*) = *initial M2*
and $\bigwedge$ *q x y q'* . *q* $\in$ *states M1* $\Longrightarrow$ *q'* $\in$ *states M1* $\Longrightarrow$ (*q*,*x*,*y*,*q'*) $\in$ *transitions M1* $\longleftrightarrow$ (*f q*,*x*,*y*,*f q'*) $\in$ *transitions M2*
and *q* $\in$ *states M1*
shows *LS M1 q* $\subseteq$ *LS M2* (*f q*)

⟨*proof*⟩

lemma *language-equivalence-from-isomorphism* :
assumes *bij-betw f (states M1) (states M2)*
and f (*initial M1*) = *initial M2*
and $\bigwedge$ *q x y q'* . *q* ∈ *states M1* ⟹ *q'* ∈ *states M1* ⟹ (*q*,*x*,*y*,*q'*) ∈ *transitions M1* ⟷ (*f q*,*x*,*y*,*f q'*) ∈ *transitions M2*
and *q* ∈ *states M1*
shows *LS M1 q* = *LS M2* (*f q*)
⟨*proof*⟩

lemma *language-equivalence-from-isomorphism-helper-reachable* :
assumes *bij-betw f (reachable-states M1) (reachable-states M2)*
and f (*initial M1*) = *initial M2*
and $\bigwedge$ *q x y q'* . *q* ∈ *reachable-states M1* ⟹ *q'* ∈ *reachable-states M1* ⟹ (*q*,*x*,*y*,*q'*) ∈ *transitions M1* ⟷ (*f q*,*x*,*y*,*f q'*) ∈ *transitions M2*
shows *L M1* ⊆ *L M2*
⟨*proof*⟩

lemma *language-equivalence-from-isomorphism-reachable* :
assumes *bij-betw f (reachable-states M1) (reachable-states M2)*
and f (*initial M1*) = *initial M2*
and $\bigwedge$ *q x y q'* . *q* ∈ *reachable-states M1* ⟹ *q'* ∈ *reachable-states M1* ⟹ (*q*,*x*,*y*,*q'*) ∈ *transitions M1* ⟷ (*f q*,*x*,*y*,*f q'*) ∈ *transitions M2*
shows *L M1* = *L M2*
⟨*proof*⟩

lemma *language-empty-io* :
assumes *inputs M* = {} ∨ *outputs M* = {}
shows *L M* = {[]}
⟨*proof*⟩

4.8 Basic FSM Properties

4.8.1 Completely Specified

fun *completely-specified* :: (*'a*,*'b*,*'c*) *fsm* ⇒ *bool* **where**
completely-specified M = (∀ *q* ∈ *states M* . ∀ *x* ∈ *inputs M* . ∃ *t* ∈ *transitions M* . *t-source t* = *q* ∧ *t-input t* = *x*)

lemma *completely-specified-alt-def* :
completely-specified M = (∀ *q* ∈ *states M* . ∀ *x* ∈ *inputs M* . ∃ *q' y* . (*q*,*x*,*y*,*q'*) ∈ *transitions M*)
⟨*proof*⟩

lemma *completely-specified-alt-def-h* :
  completely-specified M = (∀ *q* ∈ *states M* . ∀ *x* ∈ *inputs M* . *h M* (*q*,*x*) ≠ {})
  ⟨*proof*⟩

fun *completely-specified-state* :: ('*a*,'*b*,'*c*) *fsm* ⇒ '*a* ⇒ *bool* **where**
  completely-specified-state M q = (∀ *x* ∈ *inputs M* . ∃ *t* ∈ *transitions M* . *t-source t* = *q* ∧ *t-input t* = *x*)

lemma *completely-specified-states* :
  completely-specified M = (∀ *q* ∈ *states M* . *completely-specified-state M q*)
  ⟨*proof*⟩

lemma *completely-specified-state-alt-def-h* :
  completely-specified-state M q = (∀ *x* ∈ *inputs M* . *h M* (*q*,*x*) ≠ {})
  ⟨*proof*⟩

lemma *completely-specified-path-extension* :
  assumes *completely-specified M*
  and *q* ∈ *states M*
  and *path M q p*
  and *x* ∈ (*inputs M*)
obtains *t* **where** *t* ∈ *transitions M* **and** *t-input t* = *x* **and** *t-source t* = *target q p*
⟨*proof*⟩

lemma *completely-specified-language-extension* :
  assumes *completely-specified M*
  and *q* ∈ *states M*
  and *io* ∈ *LS M q*
  and *x* ∈ (*inputs M*)
obtains *y* **where** *io*@[(*x*,*y*)] ∈ *LS M q*
⟨*proof*⟩

lemma *path-of-length-ex* :
  assumes *completely-specified M*
  and *q* ∈ *states M*
  and *inputs M* ≠ {}
shows ∃ *p* . *path M q p* ∧ *length p* = *k*
⟨*proof*⟩

4.8.2 Deterministic

fun *deterministic* :: ('*a*,'*b*,'*c*) *fsm* ⇒ *bool* **where**
  deterministic M = (∀ *t1* ∈ *transitions M* .

$\forall\ t2 \in transitions\ M$.
$(t\text{-}source\ t1 = t\text{-}source\ t2 \wedge t\text{-}input\ t1 = t\text{-}input\ t2)$
$\longrightarrow (t\text{-}output\ t1 = t\text{-}output\ t2 \wedge t\text{-}target\ t1 = t\text{-}target\ t2))$

lemma *deterministic-alt-def* :
$deterministic\ M = (\forall\ q1\ x\ y'\ y''\ \ q1'\ q1''\ .\ (q1,x,y',q1') \in transitions\ M \wedge (q1,x,y'',q1'') \in transitions\ M \longrightarrow y' = y'' \wedge q1' = q1'')$
⟨*proof*⟩

lemma *deterministic-alt-def-h* :
$deterministic\ M = (\forall\ q1\ x\ yq\ yq'\ .\ (yq \in h\ M\ (q1,x) \wedge yq' \in h\ M\ (q1,x)) \longrightarrow yq = yq')$
⟨*proof*⟩

4.8.3 Observable

fun *observable* :: $('a,'b,'c)\ fsm \Rightarrow bool$ **where**
$observable\ M = (\forall\ t1 \in transitions\ M$.
$\forall\ t2 \in transitions\ M$.
$(t\text{-}source\ t1 = t\text{-}source\ t2 \wedge t\text{-}input\ t1 = t\text{-}input\ t2 \wedge t\text{-}output\ t1 = t\text{-}output\ t2)$
$\longrightarrow t\text{-}target\ t1 = t\text{-}target\ t2)$

lemma *observable-alt-def* :
$observable\ M = (\forall\ q1\ x\ y\ q1'\ q1''\ .\ (q1,x,y,q1') \in transitions\ M \wedge (q1,x,y,q1'') \in transitions\ M \longrightarrow q1' = q1'')$
⟨*proof*⟩

lemma *observable-alt-def-h* :
$observable\ M = (\forall\ q1\ x\ yq\ yq'\ .\ (yq \in h\ M\ (q1,x) \wedge yq' \in h\ M\ (q1,x)) \longrightarrow fst\ yq = fst\ yq' \longrightarrow snd\ yq = snd\ yq')$
⟨*proof*⟩

lemma *language-append-path-ob* :
assumes $io@[(x,y)] \in L\ M$
obtains $p\ t$ **where** $path\ M\ (initial\ M)\ (p@[t])$ **and** $p\text{-}io\ p = io$ **and** $t\text{-}input\ t = x$ **and** $t\text{-}output\ t = y$
⟨*proof*⟩

4.8.4 Single Input

fun *single-input* :: $('a,'b,'c)\ fsm \Rightarrow bool$ **where**
$single\text{-}input\ M = (\forall\ t1 \in transitions\ M$.
$\forall\ t2 \in transitions\ M$.
$t\text{-}source\ t1 = t\text{-}source\ t2 \longrightarrow t\text{-}input\ t1 = t\text{-}input\ t2)$

lemma *single-input-alt-def* :

single-input $M = (\forall\ q1\ x\ x'\ y\ y'\ q1'\ q1''\ .\ (q1,x,y,q1') \in$ *transitions* $M \wedge (q1,x',y',q1'') \in$ *transitions* $M \longrightarrow x = x')$
⟨*proof*⟩

lemma *single-input-alt-def-h* :
single-input $M = (\forall\ q\ x\ x'\ .\ (h\ M\ (q,x) \neq \{\} \wedge h\ M\ (q,x') \neq \{\}) \longrightarrow x = x')$
⟨*proof*⟩

4.8.5 Output Complete

fun *output-complete* :: ('a,'b,'c) *fsm* ⇒ *bool* **where**
output-complete $M = (\forall\ t \in$ *transitions* M .
$\forall\ y \in$ *outputs* M .
$\exists\ t' \in$ *transitions* M . *t-source* t = *t-source* $t' \wedge$
t-input t = *t-input* $t' \wedge$
t-output $t' = y)$

lemma *output-complete-alt-def* :
output-complete $M = (\forall\ q\ x\ .\ (\exists\ y\ q'\ .\ (q,x,y,q') \in$ *transitions* $M) \longrightarrow (\forall\ y \in ($*outputs* $M)\ .\ \exists\ q'\ .\ (q,x,y,q') \in$ *transitions* $M))$
⟨*proof*⟩

lemma *output-complete-alt-def-h* :
output-complete $M = (\forall\ q\ x\ .\ h\ M\ (q,x) \neq \{\} \longrightarrow (\forall\ y \in$ *outputs* $M\ .\ \exists\ q'\ .\ (y,q') \in h\ M\ (q,x)))$
⟨*proof*⟩

4.8.6 Acyclic

fun *acyclic* :: ('a,'b,'c) *fsm* ⇒ *bool* **where**
acyclic $M = (\forall\ p\ .$ *path* M (*initial* M) $p \longrightarrow$ *distinct* (*visited-states* (*initial* M) p))

lemma *visited-states-length* : *length* (*visited-states* q p) = *Suc* (*length* p) ⟨*proof*⟩

lemma *visited-states-take* :
(*take* (*Suc* n) (*visited-states* q p)) = (*visited-states* q (*take* n p))
⟨*proof*⟩

lemma *acyclic-code*[*code*] :
acyclic $M = (\neg(\exists\ p \in$ (*acyclic-paths-up-to-length* M (*initial* M) (*size* $M - 1$)) .
$\exists\ t \in$ *transitions* M . *t-source* t = *target* (*initial* M) $p \wedge$
t-target $t \in$ *set* (*visited-states* (*initial* M) p)))
⟨*proof*⟩

lemma *acyclic-alt-def* : *acyclic M* = *finite* (*L M*)
⟨*proof*⟩

lemma *acyclic-finite-paths-from-reachable-state* :
assumes *acyclic M*
and *path M* (*initial M*) *p*
and *target* (*initial M*) *p* = *q*
shows *finite* {*p* . *path M q p*}
⟨*proof*⟩

lemma *acyclic-paths-from-reachable-states* :
assumes *acyclic M*
and *path M* (*initial M*) *p'*
and *target* (*initial M*) *p'* = *q*
and *path M q p*
shows *distinct* (*visited-states q p*)
⟨*proof*⟩

definition *LS-acyclic* :: ('*a*,'*b*,'*c*) *fsm* ⇒ '*a* ⇒ ('*b* × '*c*) *list set* **where**
LS-acyclic M q = {*p-io p* | *p* . *path M q p* ∧ *distinct* (*visited-states q p*)}

lemma *LS-acyclic-code*[*code*] :
LS-acyclic M q = *image p-io* (*acyclic-paths-up-to-length M q* (*size M* − *1*))
⟨*proof*⟩

lemma *LS-from-LS-acyclic* :
assumes *acyclic M*
shows *L M* = *LS-acyclic M* (*initial M*)
⟨*proof*⟩

lemma *cyclic-cycle* :
assumes ¬ *acyclic M*
shows ∃ *q p* . *path M q p* ∧ *p* ≠ [] ∧ *target q p* = *q*
⟨*proof*⟩

lemma *cyclic-cycle-rev* :
fixes *M* :: ('*a*,'*b*,'*c*) *fsm*
assumes *path M* (*initial M*) *p'*
and *target* (*initial M*) *p'* = *q*
and *path M q p*
and *p* ≠ []
and *target q p* = *q*
shows ¬ *acyclic M*

⟨*proof*⟩

lemma *acyclic-initial* :
assumes *acyclic M*
shows ¬ (∃ *t* ∈ *transitions M* . *t-target t* = *initial M* ∧
(∃ *p* . *path M* (*initial M*) *p* ∧ *target* (*initial M*) *p* = *t-source t*))
⟨*proof*⟩

lemma *cyclic-path-shift* :
assumes *path M q p*
and *target q p* = *q*
shows *path M* (*target q* (*take i p*)) ((*drop i p*) @ (*take i p*))
and *target* (*target q* (*take i p*)) ((*drop i p*) @ (*take i p*)) = (*target q* (*take i p*))
⟨*proof*⟩

lemma *cyclic-path-transition-states-property* :
assumes ∃ *t* ∈ *set p* . *P* (*t-source t*)
and ∀ *t* ∈ *set p* . *P* (*t-source t*) ⟶ *P* (*t-target t*)
and *path M q p*
and *target q p* = *q*
shows ∀ *t* ∈ *set p* . *P* (*t-source t*)
and ∀ *t* ∈ *set p* . *P* (*t-target t*)
⟨*proof*⟩

lemma *cycle-incoming-transition-ex* :
assumes *path M q p*
and *p* ≠ []
and *target q p* = *q*
and *t* ∈ *set p*
shows ∃ *tI* ∈ *set p* . *t-target tI* = *t-source t*
⟨*proof*⟩

lemma *acyclic-paths-finite* :
finite {*p* . *path M q p* ∧ *distinct* (*visited-states q p*) }
⟨*proof*⟩

lemma *acyclic-no-self-loop* :
assumes *acyclic M*
and *q* ∈ *reachable-states M*
shows ¬ (∃ *x y* . (*q*,*x*,*y*,*q*) ∈ *transitions M*)
⟨*proof*⟩

4.8.7 Deadlock States

fun *deadlock-state* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $'a \Rightarrow$ *bool* **where**
  deadlock-state M q $= (\neg(\exists\ t \in$ *transitions M* . *t-source t* $= q))$

lemma *deadlock-state-alt-def* : *deadlock-state M q* $=$ (*LS M q* $\subseteq \{[]\})$
⟨*proof*⟩

lemma *deadlock-state-alt-def-h* : *deadlock-state M q* $= (\forall\ x \in$ *inputs M* . *h M* $(q,x) = \{\})$
  ⟨*proof*⟩

lemma *acyclic-deadlock-reachable* :
  assumes *acyclic M*
  shows $\exists\ q \in$ *reachable-states M* . *deadlock-state M q*
⟨*proof*⟩

lemma *deadlock-prefix* :
  assumes *path M q p*
  and $t \in$ *set* (*butlast p*)
shows $\neg$ (*deadlock-state M* (*t-target t*))
  ⟨*proof*⟩

lemma *states-initial-deadlock* :
  assumes *deadlock-state M* (*initial M*)
  shows *reachable-states M* $= \{$*initial M*$\}$

⟨*proof*⟩

4.8.8 Other

fun *completed-path* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $'a \Rightarrow$ $('a,'b,'c)$ *path* $\Rightarrow$ *bool* **where**
  completed-path M q p $=$ *deadlock-state M* (*target q p*)

fun *minimal* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ *bool* **where**
  minimal M $= (\forall\ q \in$ *states M* . $\forall\ q' \in$ *states M* . $q \neq q' \longrightarrow$ *LS M q* $\neq$ *LS M* $q')$

lemma *minimal-alt-def* : *minimal M* $= (\forall\ q\ q'$. $q \in$ *states M* $\longrightarrow q' \in$ *states M* $\longrightarrow$ *LS M q* $=$ *LS M* $q' \longrightarrow q = q')$
  ⟨*proof*⟩

definition *retains-outputs-for-states-and-inputs* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $('a,'b,'c)$ *fsm* $\Rightarrow$ *bool* **where**
  retains-outputs-for-states-and-inputs M S
    $= (\forall\ tS \in$ *transitions S* .
      $\forall\ tM \in$ *transitions M* .
        (*t-source tS* $=$ *t-source tM* $\wedge$ *t-input tS* $=$ *t-input tM*) $\longrightarrow tM \in$ *transitions*

S)

4.9 IO Targets and Observability

fun *paths-for-io′* :: $(('a \times 'b) \Rightarrow ('c \times 'a)\ set) \Rightarrow ('b \times 'c)\ list \Rightarrow 'a \Rightarrow ('a,'b,'c)$ *path* $\Rightarrow$ $('a,'b,'c)$ *path set* **where**
  paths-for-io′ f [] *q prev* = {*prev*} |
  paths-for-io′ f ((*x*,*y*)#*io*) *q prev* = ⋃(*image* (λ*yq′* . *paths-for-io′ f io* (*snd yq′*) (*prev*@[(*q*,*x*,*y*,(*snd yq′*))])) (*Set.filter* (λ*yq′* . *fst yq′* = *y*) (*f* (*q*,*x*))))

lemma *paths-for-io′-set* :
  assumes *q* ∈ *states M*
  shows *paths-for-io′* (*h M*) *io q prev* = {*prev*@*p* | *p* . *path M q p* ∧ *p-io p* = *io*}
⟨*proof*⟩

definition *paths-for-io* :: $('a,'b,'c)$ *fsm* $\Rightarrow 'a \Rightarrow ('b \times 'c)$ *list* $\Rightarrow ('a,'b,'c)$ *path set* **where**
  paths-for-io M q io = {*p* . *path M q p* ∧ *p-io p* = *io*}

lemma *paths-for-io-set-code*[*code*] :
  paths-for-io M q io = (*if q* ∈ *states M then paths-for-io′* (*h M*) *io q* [] *else* {})
  ⟨*proof*⟩

fun *io-targets* :: $('a,'b,'c)$ *fsm* $\Rightarrow ('b \times 'c)$ *list* $\Rightarrow 'a \Rightarrow 'a$ *set* **where**
  io-targets M io q = {*target q p* | *p* . *path M q p* ∧ *p-io p* = *io*}

lemma *io-targets-code*[*code*] : *io-targets M io q* = *image* (*target q*) (*paths-for-io M q io*)
  ⟨*proof*⟩

lemma *io-targets-states* :
  io-targets M io q ⊆ *states M*
  ⟨*proof*⟩

lemma *observable-transition-unique* :
  assumes *observable M*
    and *t* ∈ *transitions M*
  shows ∃! *t′* ∈ *transitions M* . *t-source t′* = *t-source t* ∧
      t-input t′ = *t-input t* ∧
      t-output t′ = *t-output t*
  ⟨*proof*⟩

lemma *observable-path-unique* :
  assumes *observable M*

and *path M q p*
and *path M q p'*
and *p-io p = p-io p'*
shows $p = p'$
⟨*proof*⟩

lemma *observable-io-targets* :
assumes *observable M*
and $io \in LS\ M\ q$
obtains q'
where *io-targets M io q* = $\{q'\}$
⟨*proof*⟩

lemma *observable-path-io-target* :
assumes *observable M*
and *path M q p*
shows *io-targets M* (*p-io p*) *q* = {*target q p*}
⟨*proof*⟩

lemma *completely-specified-io-targets* :
assumes *completely-specified M*
shows ∀ *q* ∈ *io-targets M io* (*initial M*) . ∀ *x* ∈ (*inputs M*) . ∃ *t* ∈ *transitions M* . *t-source t* = *q* ∧ *t-input t* = *x*
⟨*proof*⟩

lemma *observable-path-language-step* :
assumes *observable M*
and *path M q p*
and ¬ (∃ *t*∈*transitions M*.
t-source t = *target q p* ∧
t-input t = *x* ∧ *t-output t* = *y*)
shows (*p-io p*)@[(*x*,*y*)] ∉ *LS M q*
⟨*proof*⟩

lemma *observable-io-targets-language* :
assumes *io1* @ *io2* ∈ *LS M q1*
and *observable M*
and *q2* ∈ *io-targets M io1 q1*
shows *io2* ∈ *LS M q2*
⟨*proof*⟩

lemma *io-targets-language-append* :
assumes *q1* ∈ *io-targets M io1 q*

and *io2* ∈ *LS M q1*
shows *io1*@*io2* ∈ *LS M q*
⟨*proof*⟩

lemma *io-targets-next* :
assumes *t* ∈ *transitions M*
shows *io-targets M io* (*t-target t*) ⊆ *io-targets M* (*p-io* [*t*] @ *io*) (*t-source t*)
⟨*proof*⟩

lemma *observable-io-targets-next* :
assumes *observable M*
and *t* ∈ *transitions M*
shows *io-targets M* (*p-io* [*t*] @ *io*) (*t-source t*) = *io-targets M io* (*t-target t*)
⟨*proof*⟩

lemma *observable-language-target* :
assumes *observable M*
and *q* ∈ *io-targets M io1* (*initial M*)
and *t* ∈ *io-targets T io1* (*initial T*)
and *L T* ⊆ *L M*
shows *LS T t* ⊆ *LS M q*
⟨*proof*⟩

lemma *observable-language-target-failure* :
assumes *observable M*
and *q* ∈ *io-targets M io1* (*initial M*)
and *t* ∈ *io-targets T io1* (*initial T*)
and ¬ *LS T t* ⊆ *LS M q*
shows ¬ *L T* ⊆ *L M*
⟨*proof*⟩

lemma *language-path-append-transition-observable* :
assumes (*p-io p*) @ [(*x*,*y*)] ∈ *LS M q*
and *path M q p*
and *observable M*
obtains *t* **where** *path M q* (*p*@[*t*]) **and** *t-input t* = *x* **and** *t-output t* = *y*
⟨*proof*⟩

lemma *language-io-target-append* :
assumes *q'* ∈ *io-targets M io1 q*
and *io2* ∈ *LS M q'*
shows (*io1*@*io2*) ∈ *LS M q*

⟨*proof*⟩

lemma *observable-path-suffix* :
 assumes $(p\text{-}io\ p)@io \in LS\ M\ q$
 and $path\ M\ q\ p$
 and $observable\ M$
obtains p' **where** $path\ M\ (target\ q\ p)\ p'$ **and** $p\text{-}io\ p' = io$
⟨*proof*⟩

lemma *io-targets-finite* :
 $finite\ (io\text{-}targets\ M\ io\ q)$
⟨*proof*⟩

lemma *language-next-transition-ob* :
 assumes $(x,y)\#ios \in LS\ M\ q$
obtains t **where** $t\text{-}source\ t = q$
 and $t \in transitions\ M$
 and $t\text{-}input\ t = x$
 and $t\text{-}output\ t = y$
 and $ios \in LS\ M\ (t\text{-}target\ t)$
⟨*proof*⟩

lemma *h-observable-card* :
 assumes $observable\ M$
 shows $card\ (snd\ `\ Set.filter\ (\lambda\ (y',q')\ .\ y' = y)\ (h\ M\ (q,x))) \leq 1$
 and $finite\ (snd\ `\ Set.filter\ (\lambda\ (y',q')\ .\ y' = y)\ (h\ M\ (q,x)))$
⟨*proof*⟩

lemma *h-obs-None* :
 assumes $observable\ M$
shows $(h\text{-}obs\ M\ q\ x\ y = None) = (\nexists\, q'\ .\ (q,x,y,q') \in transitions\ M)$
⟨*proof*⟩

lemma *h-obs-Some* :
 assumes $observable\ M$
 shows $(h\text{-}obs\ M\ q\ x\ y = Some\ q') = (\{q'\ .\ (q,x,y,q') \in transitions\ M\} = \{q'\})$
⟨*proof*⟩

lemma *h-obs-state* :
 assumes $h\text{-}obs\ M\ q\ x\ y = Some\ q'$
 shows $q' \in states\ M$
⟨*proof*⟩

fun *after* :: $('a,'b,'c)\ fsm \Rightarrow 'a \Rightarrow ('b \times 'c)\ list \Rightarrow 'a$ **where**
 $after\ M\ q\ [] = q\ |$
 $after\ M\ q\ ((x,y)\#io) = after\ M\ (the\ (h\text{-}obs\ M\ q\ x\ y))\ io$

abbreviation *after-initial M io* ≡ *after M* (*initial M*) *io*

lemma *after-path* :
assumes *observable M*
and *path M q p*
shows *after M q* (*p-io p*) = *target q p*
⟨*proof*⟩

lemma *observable-after-path* :
assumes *observable M*
and *io* ∈ *LS M q*
obtains *p* **where** *path M q p*
and *p-io p* = *io*
and *target q p* = *after M q io*
⟨*proof*⟩

lemma *h-obs-from-LS* :
assumes *observable M*
and [(*x*,*y*)] ∈ *LS M q*
obtains *q'* **where** *h-obs M q x y* = *Some q'*
⟨*proof*⟩

lemma *after-h-obs* :
assumes *observable M*
and *h-obs M q x y* = *Some q'*
shows *after M q* [(*x*,*y*)] = *q'*
⟨*proof*⟩

lemma *after-h-obs-prepend* :
assumes *observable M*
and *h-obs M q x y* = *Some q'*
and *io* ∈ *LS M q'*
shows *after M q* ((*x*,*y*)#*io*) = *after M q' io*
⟨*proof*⟩

lemma *after-split* :
assumes *observable M*
and α@γ ∈ *LS M q*
shows *after M* (*after M q* α) γ = *after M q* (α @ γ)
⟨*proof*⟩

lemma *after-io-targets* :
assumes *observable M*
and *io* ∈ *LS M q*
shows *after M q io* = *the-elem* (*io-targets M io q*)
⟨*proof*⟩

lemma *after-language-subset* :
  assumes *observable M*
  and $\alpha @ \gamma \in L\ M$
  and $\beta \in LS\ M\ (\textit{after-initial}\ M\ (\alpha @ \gamma))$
shows $\gamma @ \beta \in LS\ M\ (\textit{after-initial}\ M\ \alpha)$
  ⟨*proof*⟩

lemma *after-language-append-iff* :
  assumes *observable M*
  and $\alpha @ \gamma \in L\ M$
shows $\beta \in LS\ M\ (\textit{after-initial}\ M\ (\alpha @ \gamma)) = (\gamma @ \beta \in LS\ M\ (\textit{after-initial}\ M\ \alpha))$
  ⟨*proof*⟩

lemma *h-obs-language-iff* :
  assumes *observable M*
  shows $(x,y)\#io \in LS\ M\ q = (\exists\ q'\ .\ \textit{h-obs}\ M\ q\ x\ y = \textit{Some}\ q' \wedge io \in LS\ M\ q')$
    (**is** *?P1* = *?P2*)
⟨*proof*⟩

lemma *after-language-iff* :
  assumes *observable M*
  and $\alpha \in LS\ M\ q$
shows $(\gamma \in LS\ M\ (\textit{after}\ M\ q\ \alpha)) = (\alpha @ \gamma \in LS\ M\ q)$
  ⟨*proof*⟩

lemma *language-maximal-contained-prefix-ob* :
  assumes $io \notin LS\ M\ q$
  and $q \in \textit{states}\ M$
  and *observable M*
obtains $io'\ x\ y\ io''$ **where** $io = io' @ [(x,y)] @ io''$
    and $io' \in LS\ M\ q$
    and $io' @ [(x,y)] \notin LS\ M\ q$
⟨*proof*⟩

lemma *after-is-state* :
  assumes *observable M*
  assumes $io \in LS\ M\ q$
  shows $\textit{FSM.after}\ M\ q\ io \in \textit{states}\ M$
  ⟨*proof*⟩

lemma *after-reachable-initial* :
  assumes *observable M*
  and $io \in L\ M$

shows *after-initial M io* ∈ *reachable-states M*
⟨*proof*⟩

lemma *after-transition* :
assumes *observable M*
and $(q,x,y,q') \in$ *transitions M*
shows *after M q* $[(x,y)] = q'$
⟨*proof*⟩

lemma *after-transition-exhaust* :
assumes *observable M*
and *t* ∈ *transitions M*
shows *t-target t* = *after M* (*t-source t*) [(*t-input t*, *t-output t*)]
⟨*proof*⟩

lemma *after-reachable* :
assumes *observable M*
and *io* ∈ *LS M q*
and *q* ∈ *reachable-states M*
shows *after M q io* ∈ *reachable-states M*
⟨*proof*⟩

lemma *observable-after-language-append* :
assumes *observable M*
and *io1* ∈ *LS M q*
and *io2* ∈ *LS M* (*after M q io1*)
shows *io1*@*io2* ∈ *LS M q*
⟨*proof*⟩

lemma *observable-after-language-none* :
assumes *observable M*
and *io1* ∈ *LS M q*
and *io2* ∉ *LS M* (*after M q io1*)
shows *io1*@*io2* ∉ *LS M q*
⟨*proof*⟩

lemma *observable-after-eq* :
assumes *observable M*
and *after M q io1* = *after M q io2*
and *io1* ∈ *LS M q*
and *io2* ∈ *LS M q*
shows *io1*@*io* ∈ *LS M q* ⟷ *io2*@*io* ∈ *LS M q*
⟨*proof*⟩

lemma *observable-after-target* :
assumes *observable M*
and *io* @ *io'* ∈ *LS M q*

and *path M* (*FSM.after M q io*) *p*
and *p-io p* = *io′*
shows *target* (*FSM.after M q io*) *p* = (*FSM.after M q* (*io* @ *io′*))
⟨*proof*⟩

fun *is-in-language* :: (′*a*,′*b*,′*c*) *fsm* ⇒ ′*a* ⇒ (′*b* ×′*c*) *list* ⇒ *bool* **where**
is-in-language M q [] = *True* |
is-in-language M q ((*x*,*y*)#*io*) = (*case h-obs M q x y of*
None ⇒ *False* |
Some q′ ⇒ *is-in-language M q′ io*)

lemma *is-in-language-iff* :
assumes *observable M*
and *q* ∈ *states M*
shows *is-in-language M q io* ⟷ *io* ∈ *LS M q*
⟨*proof*⟩

lemma *observable-paths-for-io* :
assumes *observable M*
and *io* ∈ *LS M q*
obtains *p* **where** *paths-for-io M q io* = {*p*}
⟨*proof*⟩

lemma *io-targets-language* :
assumes *q′* ∈ *io-targets M io q*
shows *io* ∈ *LS M q*
⟨*proof*⟩

lemma *observable-after-reachable-surj* :
assumes *observable M*
shows (*after-initial M*) ‘ (*L M*) = *reachable-states M*
⟨*proof*⟩

lemma *observable-minimal-size-r-language-distinct* :
assumes *minimal M1*
and *minimal M2*
and *observable M1*
and *observable M2*
and *size-r M1* < *size-r M2*
shows *L M1* ≠ *L M2*
⟨*proof*⟩

lemma *minimal-equivalence-size-r* :
assumes *minimal M1*
and *minimal M2*

and *observable M1*
and *observable M2*
and $L\ M1 = L\ M2$
shows *size-r M1 = size-r M2*
⟨*proof*⟩

4.10 Conformity Relations

fun *is-io-reduction-state* :: $('a,'b,'c)$ *fsm* ⇒ $'a$ ⇒ $('d,'b,'c)$ *fsm* ⇒ $'d$ ⇒ *bool* **where**
is-io-reduction-state A a B b = (*LS A a* ⊆ *LS B b*)

abbreviation(*input*) *is-io-reduction A B* ≡ *is-io-reduction-state A* (*initial A*) *B* (*initial B*)
notation
is-io-reduction (‹- ≼ -›)

fun *is-io-reduction-state-on-inputs* :: $('a,'b,'c)$ *fsm* ⇒ $'a$ ⇒ $'b$ *list set* ⇒ $('d,'b,'c)$ *fsm* ⇒ $'d$ ⇒ *bool* **where**
is-io-reduction-state-on-inputs A a U B b = (LS_{in} *A a U* ⊆ LS_{in} *B b U*)

abbreviation(*input*) *is-io-reduction-on-inputs A U B* ≡ *is-io-reduction-state-on-inputs A* (*initial A*) *U B* (*initial B*)
notation
is-io-reduction-on-inputs (‹- ≼⟦-⟧ -›)

4.11 A Pass Relation for Reduction and Test Represented as Sets of Input-Output Sequences

definition *pass-io-set* :: $('a,'b,'c)$ *fsm* ⇒ $('b \times 'c)$ *list set* ⇒ *bool* **where**
pass-io-set M ios = (∀ *io x y* . *io*@[(*x*,*y*)] ∈ *ios* ⟶ (∀ *y'* . *io*@[(*x*,*y'*)] ∈ *L M* ⟶ *io*@[(*x*,*y'*)] ∈ *ios*))

definition *pass-io-set-maximal* :: $('a,'b,'c)$ *fsm* ⇒ $('b \times 'c)$ *list set* ⇒ *bool* **where**
pass-io-set-maximal M ios = (∀ *io x y io'* . *io*@[(*x*,*y*)]@*io'* ∈ *ios* ⟶ (∀ *y'* . *io*@[(*x*,*y'*)] ∈ *L M* ⟶ (∃ *io''*. *io*@[(*x*,*y'*)]@*io''* ∈ *ios*)))

lemma *pass-io-set-from-pass-io-set-maximal* :
pass-io-set-maximal M ios = *pass-io-set M* {*io'* . ∃ *io io''* . *io* = *io'*@*io''* ∧ *io* ∈ *ios*}
⟨*proof*⟩

lemma *pass-io-set-maximal-from-pass-io-set* :
assumes *finite ios*
and ⋀ *io' io''* . *io'*@*io''* ∈ *ios* ⟹ *io'* ∈ *ios*
shows *pass-io-set M ios* = *pass-io-set-maximal M* {*io'* ∈ *ios* . ¬ (∃ *io''* . *io''* ≠ [] ∧ *io'*@*io''* ∈ *ios*)}

⟨*proof*⟩

4.12 Relaxation of IO based test suites to sets of input sequences

abbreviation(*input*) *input-portion xs* ≡ *map fst xs*

lemma *equivalence-io-relaxation* :
assumes $(L\ M1 = L\ M2) \longleftrightarrow (L\ M1 \cap T = L\ M2 \cap T)$
shows $(L\ M1 = L\ M2) \longleftrightarrow (\{io\ .\ io \in L\ M1 \wedge (\exists\ io' \in T\ .\ \text{input-portion}\ io = \text{input-portion}\ io')\} = \{io\ .\ io \in L\ M2 \wedge (\exists\ io' \in T\ .\ \text{input-portion}\ io = \text{input-portion}\ io')\})$
⟨*proof*⟩

lemma *reduction-io-relaxation* :
assumes $(L\ M1 \subseteq L\ M2) \longleftrightarrow (L\ M1 \cap T \subseteq L\ M2 \cap T)$
shows $(L\ M1 \subseteq L\ M2) \longleftrightarrow (\{io\ .\ io \in L\ M1 \wedge (\exists\ io' \in T\ .\ \text{input-portion}\ io = \text{input-portion}\ io')\} \subseteq \{io\ .\ io \in L\ M2 \wedge (\exists\ io' \in T\ .\ \text{input-portion}\ io = \text{input-portion}\ io')\})$
⟨*proof*⟩

4.13 Submachines

fun *is-submachine* :: ('a,'b,'c) *fsm* ⇒ ('a,'b,'c) *fsm* ⇒ *bool* **where**
is-submachine A B = (*initial A* = *initial B* ∧ *transitions A* ⊆ *transitions B* ∧ *inputs A* = *inputs B* ∧ *outputs A* = *outputs B* ∧ *states A* ⊆ *states B*)

lemma *submachine-path-initial* :
assumes *is-submachine A B*
and *path A* (*initial A*) *p*
shows *path B* (*initial B*) *p*
⟨*proof*⟩

lemma *submachine-path* :
assumes *is-submachine A B*
and *path A q p*
shows *path B q p*
⟨*proof*⟩

lemma *submachine-reduction* :
assumes *is-submachine A B*
shows *is-io-reduction A B*
⟨*proof*⟩

lemma *complete-submachine-initial* :

assumes *is-submachine A B*
and *completely-specified A*
shows *completely-specified-state B* (*initial B*)
⟨*proof*⟩

lemma *submachine-language* :
assumes *is-submachine S M*
shows *L S* ⊆ *L M*
⟨*proof*⟩

lemma *submachine-observable* :
assumes *is-submachine S M*
and *observable M*
shows *observable S*
⟨*proof*⟩

lemma *submachine-transitive* :
assumes *is-submachine S M*
and *is-submachine S′ S*
shows *is-submachine S′ M*
⟨*proof*⟩

lemma *transitions-subset-path* :
assumes *set p* ⊆ *transitions M*
and $p \neq []$
and *path S q p*
shows *path M q p*
⟨*proof*⟩

lemma *transition-subset-paths* :
assumes *transitions S* ⊆ *transitions M*
and *initial S* ∈ *states M*
and *inputs S* = *inputs M*
and *outputs S* = *outputs M*
and *path S* (*initial S*) *p*
shows *path M* (*initial S*) *p*
⟨*proof*⟩

lemma *submachine-reachable-subset* :
assumes *is-submachine A B*
shows *reachable-states A* ⊆ *reachable-states B*
⟨*proof*⟩

lemma *submachine-simps* :
assumes *is-submachine A B*
shows *initial A* = *initial B*
and *states A* ⊆ *states B*
and *inputs A* = *inputs B*
and *outputs A* = *outputs B*
and *transitions A* ⊆ *transitions B*
⟨*proof*⟩

lemma *submachine-deadlock* :
assumes *is-submachine A B*
and *deadlock-state B q*
shows *deadlock-state A q*
⟨*proof*⟩

4.14 Changing Initial States

lift-definition *from-FSM* :: (′*a*,′*b*,′*c*) *fsm* ⇒ ′*a* ⇒ (′*a*,′*b*,′*c*) *fsm* **is** *FSM-Impl.from-FSMI*
⟨*proof*⟩

lemma *from-FSM-simps*[*simp*]:
assumes *q* ∈ *states M*
shows
initial (*from-FSM M q*) = *q*
inputs (*from-FSM M q*) = *inputs M*
outputs (*from-FSM M q*) = *outputs M*
transitions (*from-FSM M q*) = *transitions M*
states (*from-FSM M q*) = *states M* ⟨*proof*⟩

lemma *from-FSM-path-initial* :
assumes *q* ∈ *states M*
shows *path M q p* = *path* (*from-FSM M q*) (*initial* (*from-FSM M q*)) *p*
⟨*proof*⟩

lemma *from-FSM-path* :
assumes *q* ∈ *states M*
and *path* (*from-FSM M q*) *q*′ *p*
shows *path M q*′ *p*
⟨*proof*⟩

lemma *from-FSM-reachable-states* :
assumes *q* ∈ *reachable-states M*
shows *reachable-states* (*from-FSM M q*) ⊆ *reachable-states M*
⟨*proof*⟩

lemma *submachine-from* :
assumes *is-submachine S M*
and $q \in$ *states S*
shows *is-submachine* (*from-FSM S q*) (*from-FSM M q*)
⟨*proof*⟩

lemma *from-FSM-path-rev-initial* :
assumes *path M q p*
shows *path* (*from-FSM M q*) *q p*
⟨*proof*⟩

lemma *from-from*[*simp*] :
assumes *q1* $\in$ *states M*
and *q1'* $\in$ *states M*
shows *from-FSM* (*from-FSM M q1*) *q1'* = *from-FSM M q1'* (**is** *?M* = *?M'*)
⟨*proof*⟩

lemma *from-FSM-completely-specified* :
assumes *completely-specified M*
shows *completely-specified* (*from-FSM M q*) ⟨*proof*⟩

lemma *from-FSM-single-input* :
assumes *single-input M*
shows *single-input* (*from-FSM M q*) ⟨*proof*⟩

lemma *from-FSM-acyclic* :
assumes $q \in$ *reachable-states M*
and *acyclic M*
shows *acyclic* (*from-FSM M q*)
⟨*proof*⟩

lemma *from-FSM-observable* :
assumes *observable M*
shows *observable* (*from-FSM M q*)
⟨*proof*⟩

lemma *observable-language-next* :
assumes *io#ios* $\in$ *LS M* (*t-source t*)
and *observable M*

and $t \in$ *transitions M*
and *t-input t* = *fst io*
and *t-output t* = *snd io*
shows *ios* $\in$ *L* (*from-FSM M* (*t-target t*))
⟨*proof*⟩

lemma *from-FSM-language* :
assumes *q* $\in$ *states M*
shows *L* (*from-FSM M q*) = *LS M q*
⟨*proof*⟩

lemma *observable-transition-target-language-subset* :
assumes *LS M* (*t-source t1*) $\subseteq$ *LS M* (*t-source t2*)
and *t1* $\in$ *transitions M*
and *t2* $\in$ *transitions M*
and *t-input t1* = *t-input t2*
and *t-output t1* = *t-output t2*
and *observable M*
shows *LS M* (*t-target t1*) $\subseteq$ *LS M* (*t-target t2*)
⟨*proof*⟩

lemma *observable-transition-target-language-eq* :
assumes *LS M* (*t-source t1*) = *LS M* (*t-source t2*)
and *t1* $\in$ *transitions M*
and *t2* $\in$ *transitions M*
and *t-input t1* = *t-input t2*
and *t-output t1* = *t-output t2*
and *observable M*
shows *LS M* (*t-target t1*) = *LS M* (*t-target t2*)
⟨*proof*⟩

lemma *language-state-prepend-transition* :
assumes *io* $\in$ *LS* (*from-FSM A* (*t-target t*)) (*initial* (*from-FSM A* (*t-target t*)))
and *t* $\in$ *transitions A*
shows *p-io* [*t*] @ *io* $\in$ *LS A* (*t-source t*)
⟨*proof*⟩

lemma *observable-language-transition-target* :
assumes *observable M*
and *t* $\in$ *transitions M*
and (*t-input t*, *t-output t*) # *io* $\in$ *LS M* (*t-source t*)
shows *io* $\in$ *LS M* (*t-target t*)
⟨*proof*⟩

lemma *LS-single-transition* :
[(*x*,*y*)] $\in$ *LS M q* $\longleftrightarrow$ ($\exists$ *t* $\in$ *transitions M* . *t-source t* = *q* $\wedge$ *t-input t* = *x* $\wedge$

t-output t = y)
⟨*proof*⟩

lemma *h-obs-language-append* :
assumes *observable M*
and $u \in L\ M$
and *h-obs M* (*after-initial M u*) *x y* ≠ *None*
shows $u@[(x,y)] \in L\ M$
⟨*proof*⟩

lemma *h-obs-language-single-transition-iff* :
assumes *observable M*
shows $[(x,y)] \in LS\ M\ q \longleftrightarrow$ *h-obs M q x y* ≠ *None*
⟨*proof*⟩

lemma *minimal-failure-prefix-ob* :
assumes *observable M*
and *observable I*
and $qM \in states\ M$
and $qI \in states\ I$
and $io \in LS\ I\ qI - LS\ M\ qM$
obtains $io'\ xy\ io''$ **where** $io = io'@[xy]@io''$
and $io' \in LS\ I\ qI \cap LS\ M\ qM$
and $io'@[xy] \in LS\ I\ qI - LS\ M\ qM$
⟨*proof*⟩

4.15 Language and Defined Inputs

lemma *defined-inputs-code* : *defined-inputs M q* = *t-input* ‘ *Set.filter* (λ*t* . *t-source t* = *q*) (*transitions M*)
⟨*proof*⟩

lemma *defined-inputs-alt-def* : *defined-inputs M q* = {*t-input t* | *t* . *t* ∈ *transitions M* ∧ *t-source t* = *q*}
⟨*proof*⟩

lemma *defined-inputs-language-diff* :
assumes *x* ∈ *defined-inputs M1 q1*
and *x* ∉ *defined-inputs M2 q2*
obtains *y* **where** $[(x,y)] \in LS\ M1\ q1 - LS\ M2\ q2$
⟨*proof*⟩

lemma *language-path-append* :
assumes *path M1 q1 p1*
and *io* ∈ *LS M1* (*target q1 p1*)
shows (*p-io p1* @ *io*) ∈ *LS M1 q1*
⟨*proof*⟩

lemma *observable-defined-inputs-diff-ob* :
assumes *observable M1*
and *observable M2*
and *path M1 q1 p1*
and *path M2 q2 p2*
and *p-io p1 = p-io p2*
and $x \in$ *defined-inputs M1* (*target q1 p1*)
and $x \notin$ *defined-inputs M2* (*target q2 p2*)
obtains *y* **where** (*p-io p1*)@[(*x*,*y*)] $\in$ *LS M1 q1* $-$ *LS M2 q2*
⟨*proof*⟩

lemma *observable-defined-inputs-diff-language* :
assumes *observable M1*
and *observable M2*
and *path M1 q1 p1*
and *path M2 q2 p2*
and *p-io p1 = p-io p2*
and *defined-inputs M1* (*target q1 p1*) $\neq$ *defined-inputs M2* (*target q2 p2*)
shows *LS M1 q1* $\neq$ *LS M2 q2*
⟨*proof*⟩

fun *maximal-prefix-in-language* :: ('*a*,'*b*,'*c*) *fsm* ⇒ '*a* ⇒ ('*b* ×'*c*) *list* ⇒ ('*b* ×'*c*) *list* **where**
maximal-prefix-in-language M q [] = [] |
maximal-prefix-in-language M q ((*x*,*y*)#*io*) = (*case h-obs M q x y of*
None ⇒ [] |
Some q' ⇒ (*x*,*y*)#*maximal-prefix-in-language M q' io*)

lemma *maximal-prefix-in-language-properties* :
assumes *observable M*
and $q \in$ *states M*
shows *maximal-prefix-in-language M q io* $\in$ *LS M q*
and *maximal-prefix-in-language M q io* $\in$ *list.set* (*prefixes io*)
⟨*proof*⟩

4.16 Further Reachability Formalisations

fun *reachable-k* :: ('*a*,'*b*,'*c*) *fsm* ⇒ '*a* ⇒ *nat* ⇒ '*a set* **where**
reachable-k M q n = {*target q p* | *p* . *path M q p* ∧ *length p* ≤ *n*}

lemma *reachable-k-0-initial* : *reachable-k M* (*initial M*) *0* = {*initial M*}
⟨*proof*⟩

lemma *reachable-k-states* : *reachable-states M* = *reachable-k M* (*initial M*) (*size M* − *1*)
⟨*proof*⟩

4.16.1 Induction Schemes

lemma *acyclic-induction* [*consumes 1*, *case-names reachable-state*]:
assumes *acyclic M*
and $\bigwedge$ *q . q* ∈ *reachable-states M* ⟹ ($\bigwedge$ *t . t* ∈ *transitions M* ⟹ ((*t-source t* = *q*) ⟹ *P* (*t-target t*))) ⟹ *P q*
shows ∀ *q* ∈ *reachable-states M . P q*
⟨*proof*⟩

lemma *reachable-states-induct* [*consumes 1*, *case-names init transition*] :
assumes *q* ∈ *reachable-states M*
and *P* (*initial M*)
and $\bigwedge$ *t . t* ∈ *transitions M* ⟹ *t-source t* ∈ *reachable-states M* ⟹ *P* (*t-source t*) ⟹ *P* (*t-target t*)
shows *P q*
⟨*proof*⟩

lemma *reachable-states-cases* [*consumes 1*, *case-names init transition*] :
assumes *q* ∈ *reachable-states M*
and *P* (*initial M*)
and $\bigwedge$ *t . t* ∈ *transitions M* ⟹ *t-source t* ∈ *reachable-states M* ⟹ *P* (*t-target t*)
shows *P q*
⟨*proof*⟩

4.17 Further Path Enumeration Algorithms

fun *paths-for-input′* :: (′*a* ⇒ (′*b* × ′*c* × ′*a*) *set*) ⇒ ′*b list* ⇒ ′*a* ⇒ (′*a*,′*b*,′*c*) *path* ⇒ (′*a*,′*b*,′*c*) *path set* **where**
paths-for-input′ f [] *q prev* = {*prev*} |
paths-for-input′ f (*x*#*xs*) *q prev* = ⋃(*image* (λ(*x′*,*y′*,*q′*) . *paths-for-input′ f xs q′* (*prev*@[(*q*,*x*,*y′*,*q′*)])) (*Set.filter* (λ(*x′*,*y′*,*q′*) . *x′* = *x*) (*f q*)))

lemma *paths-for-input′-set* :
assumes *q* ∈ *states M*
shows *paths-for-input′* (*h-from M*) *xs q prev* = {*prev*@*p* | *p* . *path M q p* ∧ *map fst* (*p-io p*) = *xs*}
⟨*proof*⟩

definition *paths-for-input* :: (′*a*,′*b*,′*c*) *fsm* ⇒ ′*a* ⇒ ′*b list* ⇒ (′*a*,′*b*,′*c*) *path set* **where**
paths-for-input M q xs = {*p* . *path M q p* ∧ *map fst* (*p-io p*) = *xs*}

lemma *paths-for-input-set-code*[*code*] :
paths-for-input M q xs = (*if q* ∈ *states M then paths-for-input′* (*h-from M*) *xs q* [] *else* {})
⟨*proof*⟩

fun *paths-up-to-length-or-condition-with-witness'* ::
('a ⇒ ('b × 'c × 'a) set) ⇒ (('a,'b,'c) path ⇒ 'd option) ⇒ ('a,'b,'c) path ⇒ nat ⇒ 'a ⇒ (('a,'b,'c) path × 'd) set
where
paths-up-to-length-or-condition-with-witness' f P prev 0 q = (*case P prev of Some w* ⇒ {(*prev*,*w*)} | *None* ⇒ {}) |
paths-up-to-length-or-condition-with-witness' f P prev (*Suc k*) *q* = (*case P prev of*
Some w ⇒ {(*prev*,*w*)} |
None ⇒ (⋃(*image* (λ(*x*,*y*,*q'*) . *paths-up-to-length-or-condition-with-witness' f P* (*prev*@[(*q*,*x*,*y*,*q'*)]) *k q'*) (*f q*))))

lemma *paths-up-to-length-or-condition-with-witness'-set* :
assumes *q* ∈ *states M*
shows *paths-up-to-length-or-condition-with-witness'* (*h-from M*) *P prev k q*
= {(*prev*@*p*,*x*) | *p x* . *path M q p*
∧ *length p* ≤ *k*
∧ *P* (*prev*@*p*) = *Some x*
∧ (∀ *p' p''* . (*p* = *p'*@*p''* ∧ *p''* ≠ []) ⟶ *P* (*prev*@*p'*) = *None*)}
⟨*proof*⟩

definition *paths-up-to-length-or-condition-with-witness* ::
('a,'b,'c) *fsm* ⇒ (('a,'b,'c) *path* ⇒ 'd *option*) ⇒ *nat* ⇒ 'a ⇒ (('a,'b,'c) *path* × 'd) *set*
where
paths-up-to-length-or-condition-with-witness M P k q
= {(*p*,*x*) | *p x* . *path M q p*
∧ *length p* ≤ *k*
∧ *P p* = *Some x*
∧ (∀ *p' p''* . (*p* = *p'*@*p''* ∧ *p''* ≠ []) ⟶ *P p'* = *None*)}

lemma *paths-up-to-length-or-condition-with-witness-code*[*code*] :
paths-up-to-length-or-condition-with-witness M P k q
= (*if q* ∈ *states M then paths-up-to-length-or-condition-with-witness'* (*h-from M*) *P* [] *k q*
else {})
⟨*proof*⟩

lemma *paths-up-to-length-or-condition-with-witness-finite* :
finite (*paths-up-to-length-or-condition-with-witness M P k q*)
⟨*proof*⟩

4.18 More Acyclicity Properties

lemma *maximal-path-target-deadlock* :
assumes *path M* (*initial M*) *p*
and $\neg(\exists\ p'$. *path M* (*initial M*) $p' \wedge$ *is-prefix p* $p' \wedge p \neq p')$
shows *deadlock-state M* (*target* (*initial M*) *p*)
⟨*proof*⟩

lemma *path-to-deadlock-is-maximal* :
assumes *path M* (*initial M*) *p*
and *deadlock-state M* (*target* (*initial M*) *p*)
shows $\neg(\exists\ p'$. *path M* (*initial M*) $p' \wedge$ *is-prefix p* $p' \wedge p \neq p')$
⟨*proof*⟩

definition *maximal-acyclic-paths* :: ('a,'b,'c) *fsm* ⇒ ('a,'b,'c) *path set* **where**
maximal-acyclic-paths M = {*p* . *path M* (*initial M*) *p*
∧ *distinct* (*visited-states* (*initial M*) *p*)
∧ ¬(∃ *p'* . *p'* ≠ [] ∧ *path M* (*initial M*) (*p*@*p'*)
∧ *distinct* (*visited-states* (*initial M*) (*p*@*p'*)))}

lemma *maximal-acyclic-paths-code*[*code*] :
maximal-acyclic-paths M = (*let ps* = *acyclic-paths-up-to-length M* (*initial M*) (*size M* − *1*)
in Set.filter (λ*p* . ¬ (∃ *p'* ∈ *ps* . *p'* ≠ *p* ∧ *is-prefix p p'*)) *ps*)
⟨*proof*⟩

lemma *maximal-acyclic-path-deadlock* :
assumes *acyclic M*
and *path M* (*initial M*) *p*
shows ¬(∃ *p'* . *p'* ≠ [] ∧ *path M* (*initial M*) (*p*@*p'*) ∧ *distinct* (*visited-states* (*initial M*) (*p*@*p'*)))
= *deadlock-state M* (*target* (*initial M*) *p*)
⟨*proof*⟩

lemma *maximal-acyclic-paths-deadlock-targets* :
assumes *acyclic M*
shows *maximal-acyclic-paths M*
= { *p* . *path M* (*initial M*) *p* ∧ *deadlock-state M* (*target* (*initial M*) *p*)}
⟨*proof*⟩

lemma *cycle-from-cyclic-path* :

assumes *path M q p*
and ¬ *distinct* (*visited-states q p*)
obtains *i j* **where**
take j (*drop i p*) ≠ []
target (*target q* (*take i p*)) (*take j* (*drop i p*)) = (*target q* (*take i p*))
path M (*target q* (*take i p*)) (*take j* (*drop i p*))
⟨*proof*⟩

lemma *acyclic-single-deadlock-reachable* :
assumes *acyclic M*
and ⋀ *q'* . *q'* ∈ *reachable-states M* ⟹ *q'* = *qd* ∨ ¬ *deadlock-state M q'*
shows *qd* ∈ *reachable-states M*
⟨*proof*⟩

lemma *acyclic-paths-to-single-deadlock* :
assumes *acyclic M*
and ⋀ *q'* . *q'* ∈ *reachable-states M* ⟹ *q'* = *qd* ∨ ¬ *deadlock-state M q'*
and *q* ∈ *reachable-states M*
obtains *p* **where** *path M q p* **and** *target q p* = *qd*
⟨*proof*⟩

4.19 Elements as Lists

fun *states-as-list* :: (*'a* :: *linorder*, *'b*, *'c*) *fsm* ⇒ *'a list* **where**
states-as-list M = *sorted-list-of-set* (*states M*)

lemma *states-as-list-distinct* : *distinct* (*states-as-list M*) ⟨*proof*⟩

lemma *states-as-list-set* : *set* (*states-as-list M*) = *states M*
⟨*proof*⟩

fun *reachable-states-as-list* :: (*'a* :: *linorder*, *'b*, *'c*) *fsm* ⇒ *'a list* **where**
reachable-states-as-list M = *sorted-list-of-set* (*reachable-states M*)

lemma *reachable-states-as-list-distinct* : *distinct* (*reachable-states-as-list M*) ⟨*proof*⟩

lemma *reachable-states-as-list-set* : *set* (*reachable-states-as-list M*) = *reachable-states M*
⟨*proof*⟩

fun *inputs-as-list* :: (*'a*, *'b* :: *linorder*, *'c*) *fsm* ⇒ *'b list* **where**
inputs-as-list M = *sorted-list-of-set* (*inputs M*)

lemma *inputs-as-list-set* : *set* (*inputs-as-list M*) = *inputs M*

⟨*proof*⟩

lemma *inputs-as-list-distinct* : *distinct* (*inputs-as-list M*) ⟨*proof*⟩

fun *transitions-as-list* :: ('*a* :: *linorder*,'*b* :: *linorder*,'*c* :: *linorder*) *fsm* ⇒ ('*a*,'*b*,'*c*) *transition list* **where**
transitions-as-list M = *sorted-list-of-set* (*transitions M*)

lemma *transitions-as-list-set* : *set* (*transitions-as-list M*) = *transitions M*
⟨*proof*⟩

fun *outputs-as-list* :: ('*a*,'*b*,'*c* :: *linorder*) *fsm* ⇒ '*c list* **where**
outputs-as-list M = *sorted-list-of-set* (*outputs M*)

lemma *outputs-as-list-set* : *set* (*outputs-as-list M*) = *outputs M*
⟨*proof*⟩

fun *ftransitions* :: ('*a* :: *linorder*,'*b* :: *linorder*,'*c* :: *linorder*) *fsm* ⇒ ('*a*,'*b*,'*c*) *transition fset* **where**
ftransitions M = *fset-of-list* (*transitions-as-list M*)

fun *fstates* :: ('*a* :: *linorder*,'*b*,'*c*) *fsm* ⇒ '*a fset* **where**
fstates M = *fset-of-list* (*states-as-list M*)

fun *finputs* :: ('*a*,'*b* :: *linorder*,'*c*) *fsm* ⇒ '*b fset* **where**
finputs M = *fset-of-list* (*inputs-as-list M*)

fun *foutputs* :: ('*a*,'*b*,'*c* :: *linorder*) *fsm* ⇒ '*c fset* **where**
foutputs M = *fset-of-list* (*outputs-as-list M*)

lemma *fstates-set* : *fset* (*fstates M*) = *states M*
⟨*proof*⟩

lemma *finputs-set* : *fset* (*finputs M*) = *inputs M*
⟨*proof*⟩

lemma *foutputs-set* : *fset* (*foutputs M*) = *outputs M*
⟨*proof*⟩

lemma *ftransitions-set*: *fset* (*ftransitions M*) = *transitions M*
⟨*proof*⟩

lemma *ftransitions-source*:
q |∈| (*t-source* |'| *ftransitions M*) ⟹ *q* ∈ *states M*
⟨*proof*⟩

lemma *ftransitions-target*:
q |∈| (*t-target* |'| *ftransitions M*) ⟹ *q* ∈ *states M*
⟨*proof*⟩

4.20 Responses to Input Sequences

fun *language-for-input* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ '*a* ⇒ '*b list* ⇒ ('*b*×'*c*) *list list* **where**
language-for-input M q [] = [[]] |
language-for-input M q (*x*#*xs*) =
(*let outs* = *outputs-as-list M*
in concat (*map* (λ*y* . *case h-obs M q x y of None* ⇒ [] | *Some q'* ⇒ *map* ((#) (*x*,*y*)) (*language-for-input M q' xs*)) *outs*))

lemma *language-for-input-set* :
assumes *observable M*
and *q* ∈ *states M*
shows *list.set* (*language-for-input M q xs*) = {*io* . *io* ∈ *LS M q* ∧ *map fst io* = *xs*}
⟨*proof*⟩

4.21 Filtering Transitions

lift-definition *filter-transitions* ::
('*a*,'*b*,'*c*) *fsm* ⇒ (('*a*,'*b*,'*c*) *transition* ⇒ *bool*) ⇒ ('*a*,'*b*,'*c*) *fsm* **is** *FSM-Impl.filter-transitions*

⟨*proof*⟩

lemma *filter-transitions-simps*[*simp*] :
initial (*filter-transitions M P*) = *initial M*
states (*filter-transitions M P*) = *states M*
inputs (*filter-transitions M P*) = *inputs M*
outputs (*filter-transitions M P*) = *outputs M*
transitions (*filter-transitions M P*) = {*t* ∈ *transitions M* . *P t*}
⟨*proof*⟩

lemma *filter-transitions-submachine* :
is-submachine (*filter-transitions M P*) *M*
⟨*proof*⟩

lemma *filter-transitions-path* :
assumes *path* (*filter-transitions M P*) *q p*
shows *path M q p*
⟨*proof*⟩

lemma *filter-transitions-reachable-states* :
assumes *q* ∈ *reachable-states* (*filter-transitions M P*)
shows *q* ∈ *reachable-states M*
⟨*proof*⟩

4.22 Filtering States

lift-definition *filter-states* :: ('a,'b,'c) *fsm* ⇒ ('a ⇒ *bool*) ⇒ ('a,'b,'c) *fsm*
is *FSM-Impl.filter-states*
⟨*proof*⟩

lemma *filter-states-simps*[*simp*] :
assumes *P* (*initial M*)
shows *initial* (*filter-states M P*) = *initial M*
states (*filter-states M P*) = *Set.filter P* (*states M*)
inputs (*filter-states M P*) = *inputs M*
outputs (*filter-states M P*) = *outputs M*
transitions (*filter-states M P*) = {*t* ∈ *transitions M* . *P* (*t-source t*) ∧ *P* (*t-target t*)}
⟨*proof*⟩

lemma *filter-states-submachine* :
assumes *P* (*initial M*)
shows *is-submachine* (*filter-states M P*) *M*
⟨*proof*⟩

fun *restrict-to-reachable-states* :: ('a,'b,'c) *fsm* ⇒ ('a,'b,'c) *fsm* **where**
restrict-to-reachable-states M = *filter-states M* (λ *q* . *q* ∈ *reachable-states M*)

lemma *restrict-to-reachable-states-simps*[*simp*] :
shows *initial* (*restrict-to-reachable-states M*) = *initial M*
states (*restrict-to-reachable-states M*) = *reachable-states M*
inputs (*restrict-to-reachable-states M*) = *inputs M*
outputs (*restrict-to-reachable-states M*) = *outputs M*
transitions (*restrict-to-reachable-states M*)
= {*t* ∈ *transitions M* . (*t-source t*) ∈ *reachable-states M*}
⟨*proof*⟩

lemma *restrict-to-reachable-states-path* :
assumes *q* ∈ *reachable-states M*
shows *path M q p* = *path* (*restrict-to-reachable-states M*) *q p*
⟨*proof*⟩

lemma *restrict-to-reachable-states-language* :
L (*restrict-to-reachable-states M*) = *L M*
⟨*proof*⟩

lemma *restrict-to-reachable-states-observable* :
assumes *observable M*
shows *observable* (*restrict-to-reachable-states M*)

⟨*proof*⟩

lemma *restrict-to-reachable-states-minimal* :
assumes *minimal M*
shows *minimal* (*restrict-to-reachable-states M*)
⟨*proof*⟩

lemma *restrict-to-reachable-states-reachable-states* :
reachable-states (*restrict-to-reachable-states M*) = *states* (*restrict-to-reachable-states M*)
⟨*proof*⟩

4.23 Adding Transitions

lift-definition *create-unconnected-fsm* :: *'a ⇒ 'a set ⇒ 'b set ⇒ 'c set ⇒ ('a,'b,'c) fsm*
is *FSM-Impl.create-unconnected-FSMI* ⟨*proof*⟩

lemma *create-unconnected-fsm-simps* :
assumes *finite ns* **and** *finite ins* **and** *finite outs* **and** *q ∈ ns*
shows *initial* (*create-unconnected-fsm q ns ins outs*) = *q*
states (*create-unconnected-fsm q ns ins outs*) = *ns*
inputs (*create-unconnected-fsm q ns ins outs*) = *ins*
outputs (*create-unconnected-fsm q ns ins outs*) = *outs*
transitions (*create-unconnected-fsm q ns ins outs*) = {}
⟨*proof*⟩

lift-definition *create-unconnected-fsm-from-lists* :: *'a ⇒ 'a list ⇒ 'b list ⇒ 'c list ⇒ ('a,'b,'c) fsm*
is *FSM-Impl.create-unconnected-fsm-from-lists* ⟨*proof*⟩

lemma *create-unconnected-fsm-from-lists-simps* :
assumes *q ∈ set ns*
shows *initial* (*create-unconnected-fsm-from-lists q ns ins outs*) = *q*
states (*create-unconnected-fsm-from-lists q ns ins outs*) = *set ns*
inputs (*create-unconnected-fsm-from-lists q ns ins outs*) = *set ins*
outputs (*create-unconnected-fsm-from-lists q ns ins outs*) = *set outs*
transitions (*create-unconnected-fsm-from-lists q ns ins outs*) = {}
⟨*proof*⟩

lift-definition *create-unconnected-fsm-from-fsets* :: *'a ⇒ 'a fset ⇒ 'b fset ⇒ 'c fset ⇒ ('a,'b,'c) fsm*
is *FSM-Impl.create-unconnected-fsm-from-fsets* ⟨*proof*⟩

lemma *create-unconnected-fsm-from-fsets-simps* :
assumes *q |∈| ns*
shows *initial* (*create-unconnected-fsm-from-fsets q ns ins outs*) = *q*
states (*create-unconnected-fsm-from-fsets q ns ins outs*) = *fset ns*
inputs (*create-unconnected-fsm-from-fsets q ns ins outs*) = *fset ins*

outputs (*create-unconnected-fsm-from-fsets q ns ins outs*) = *fset outs*
transitions (*create-unconnected-fsm-from-fsets q ns ins outs*) = {}
⟨*proof*⟩

lift-definition *add-transitions* :: ($'a,'b,'c$) *fsm* ⇒ ($'a,'b,'c$) *transition set* ⇒ ($'a,'b,'c$) *fsm*
is *FSM-Impl.add-transitions*
⟨*proof*⟩

lemma *add-transitions-simps* :
assumes ⋀ *t* . *t* ∈ *ts* ⟹ *t-source t* ∈ *states M* ∧ *t-input t* ∈ *inputs M* ∧ *t-output t* ∈ *outputs M* ∧ *t-target t* ∈ *states M*
shows *initial* (*add-transitions M ts*) = *initial M*
states (*add-transitions M ts*) = *states M*
inputs (*add-transitions M ts*) = *inputs M*
outputs (*add-transitions M ts*) = *outputs M*
transitions (*add-transitions M ts*) = *transitions M* ∪ *ts*
⟨*proof*⟩

lift-definition *create-fsm-from-sets* :: $'a$ ⇒ $'a$ *set* ⇒ $'b$ *set* ⇒ $'c$ *set* ⇒ ($'a,'b,'c$) *transition set* ⇒ ($'a,'b,'c$) *fsm*
is *FSM-Impl.create-fsm-from-sets*
⟨*proof*⟩

lemma *create-fsm-from-sets-simps* :
assumes *q* ∈ *qs* **and** *finite qs* **and** *finite ins* **and** *finite outs*
assumes ⋀ *t* . *t* ∈ *ts* ⟹ *t-source t* ∈ *qs* ∧ *t-input t* ∈ *ins* ∧ *t-output t* ∈ *outs* ∧ *t-target t* ∈ *qs*
shows *initial* (*create-fsm-from-sets q qs ins outs ts*) = *q*
states (*create-fsm-from-sets q qs ins outs ts*) = *qs*
inputs (*create-fsm-from-sets q qs ins outs ts*) = *ins*
outputs (*create-fsm-from-sets q qs ins outs ts*) = *outs*
transitions (*create-fsm-from-sets q qs ins outs ts*) = *ts*
⟨*proof*⟩

lemma *create-fsm-from-self* :
m = *create-fsm-from-sets* (*initial m*) (*states m*) (*inputs m*) (*outputs m*) (*transitions m*)
⟨*proof*⟩

lemma *create-fsm-from-sets-surj* :
assumes *finite* (*UNIV* :: $'a$ *set*)
and *finite* (*UNIV* :: $'b$ *set*)
and *finite* (*UNIV* :: $'c$ *set*)
shows *surj* (λ(*q*::$'a$,*Q*,*X*::$'b$ *set*,*Y*::$'c$ *set*,*T*) . *create-fsm-from-sets q Q X Y T*)

⟨*proof*⟩

4.24 Distinguishability

definition *distinguishes* :: ('a,'b,'c) *fsm* ⇒ 'a ⇒ 'a ⇒ ('b × 'c) *list* ⇒ *bool* **where**
distinguishes M q1 q2 io = (*io* ∈ *LS M q1* ∪ *LS M q2* ∧ *io* ∉ *LS M q1* ∩ *LS M q2*)

definition *minimally-distinguishes* :: ('a,'b,'c) *fsm* ⇒ 'a ⇒ 'a ⇒ ('b × 'c) *list* ⇒ *bool* **where**
minimally-distinguishes M q1 q2 io = (*distinguishes M q1 q2 io*
∧ (∀ *io'* . *distinguishes M q1 q2 io'* ⟶ *length io* ≤ *length io'*))

lemma *minimally-distinguishes-ex* :
assumes *q1* ∈ *states M*
and *q2* ∈ *states M*
and *LS M q1* ≠ *LS M q2*
obtains *v* **where** *minimally-distinguishes M q1 q2 v*
⟨*proof*⟩

lemma *distinguish-prepend* :
assumes *observable M*
and *distinguishes M* (*FSM.after M q1 io*) (*FSM.after M q2 io*) *w*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *io* ∈ *LS M q1*
and *io* ∈ *LS M q2*
shows *distinguishes M q1 q2* (*io*@*w*)
⟨*proof*⟩

lemma *distinguish-prepend-initial* :
assumes *observable M*
and *distinguishes M* (*after-initial M* (*io1*@*io*)) (*after-initial M* (*io2*@*io*)) *w*
and *io1*@*io* ∈ *L M*
and *io2*@*io* ∈ *L M*
shows *distinguishes M* (*after-initial M io1*) (*after-initial M io2*) (*io*@*w*)
⟨*proof*⟩

lemma *minimally-distinguishes-no-prefix* :
assumes *observable M*
and *u*@*w* ∈ *L M*
and *v*@*w* ∈ *L M*
and *minimally-distinguishes M* (*after-initial M u*) (*after-initial M v*) (*w*@*w'*@*w''*)
and *w'* ≠ []
shows ¬*distinguishes M* (*after-initial M* (*u*@*w*)) (*after-initial M* (*v*@*w*)) *w''*
⟨*proof*⟩

lemma *minimally-distinguishes-after-append* :
assumes *observable M*
and *minimal M*
and $q1 \in states\ M$
and $q2 \in states\ M$
and *minimally-distinguishes M q1 q2* $(w@w')$
and $w' \neq []$
shows *minimally-distinguishes M* (*after M q1 w*) (*after M q2 w*) w'
⟨*proof*⟩

lemma *minimally-distinguishes-after-append-initial* :
assumes *observable M*
and *minimal M*
and $u \in L\ M$
and $v \in L\ M$
and *minimally-distinguishes M* (*after-initial M u*) (*after-initial M v*) $(w@w')$
and $w' \neq []$
shows *minimally-distinguishes M* (*after-initial M* $(u@w)$) (*after-initial M* $(v@w)$) w'
⟨*proof*⟩

lemma *minimally-distinguishes-proper-prefixes-card* :
assumes *observable M*
and *minimal M*
and $q1 \in states\ M$
and $q2 \in states\ M$
and *minimally-distinguishes M q1 q2 w*
and $S \subseteq states\ M$
shows $card\ \{w'\ .\ w' \in set\ (prefixes\ w) \wedge w' \neq w \wedge after\ M\ q1\ w' \in S \wedge after\ M\ q2\ w' \in S\} \leq card\ S - 1$
(**is** *?P S*)
⟨*proof*⟩

lemma *minimally-distinguishes-proper-prefix-in-language* :
assumes *minimally-distinguishes M q1 q2 io*
and $io' \in set\ (prefixes\ io)$
and $io' \neq io$
shows $io' \in LS\ M\ q1 \cap LS\ M\ q2$
⟨*proof*⟩

lemma *distinguishes-not-Nil*:
assumes *distinguishes M q1 q2 io*
and $q1 \in states\ M$
and $q2 \in states\ M$
shows $io \neq []$

⟨*proof*⟩

fun *does-distinguish* :: ('a,'b,'c) *fsm* ⇒ 'a ⇒ 'a ⇒ ('b × 'c) *list* ⇒ *bool* **where**
does-distinguish M q1 q2 io = (*is-in-language M q1 io* ≠ *is-in-language M q2 io*)

lemma *does-distinguish-correctness* :
assumes *observable M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *does-distinguish M q1 q2 io* = *distinguishes M q1 q2 io*
⟨*proof*⟩

lemma *h-obs-distinguishes* :
assumes *observable M*
and *h-obs M q1 x y* = *Some q1'*
and *h-obs M q2 x y* = *None*
shows *distinguishes M q1 q2* [(*x*,*y*)]
⟨*proof*⟩

lemma *distinguishes-sym* :
assumes *distinguishes M q1 q2 io*
shows *distinguishes M q2 q1 io*
⟨*proof*⟩

lemma *distinguishes-after-prepend* :
assumes *observable M*
and *h-obs M q1 x y* ≠ *None*
and *h-obs M q2 x y* ≠ *None*
and *distinguishes M* (*FSM.after M q1* [(*x*,*y*)]) (*FSM.after M q2* [(*x*,*y*)]) γ
shows *distinguishes M q1 q2* ((*x*,*y*)#γ)
⟨*proof*⟩

lemma *distinguishes-after-initial-prepend* :
assumes *observable M*
and *io1* ∈ *L M*
and *io2* ∈ *L M*
and *h-obs M* (*after-initial M io1*) *x y* ≠ *None*
and *h-obs M* (*after-initial M io2*) *x y* ≠ *None*
and *distinguishes M* (*after-initial M* (*io1*@[(*x*,*y*)])) (*after-initial M* (*io2*@[(*x*,*y*)])) γ
shows *distinguishes M* (*after-initial M io1*) (*after-initial M io2*) ((*x*,*y*)#γ)
⟨*proof*⟩

4.25 Extending FSMs by single elements

lemma *fsm-from-list-simps*[*simp*] :
initial (*fsm-from-list q ts*) = (*case ts of* [] ⇒ *q* | (*t*#*ts*) ⇒ *t-source t*)
states (*fsm-from-list q ts*) = (*case ts of* [] ⇒ {*q*} | (*t*#*ts'*) ⇒ ((*image t-source* (*set ts*)) ∪ (*image t-target* (*set ts*))))

inputs (*fsm-from-list q ts*) = *image t-input* (*set ts*)
outputs (*fsm-from-list q ts*) = *image t-output* (*set ts*)
transitions (*fsm-from-list q ts*) = *set ts*
⟨*proof*⟩

lift-definition *add-transition* :: ($'a,'b,'c$) *fsm* ⇒ ($'a,'b,'c$) *transition* ⇒ ($'a,'b,'c$) *fsm* **is** *FSM-Impl.add-transition*
⟨*proof*⟩

lemma *add-transition-simps*[*simp*]:
assumes *t-source t* ∈ *states M* **and** *t-input t* ∈ *inputs M* **and** *t-output t* ∈ *outputs M* **and** *t-target t* ∈ *states M*
shows
initial (*add-transition M t*) = *initial M*
inputs (*add-transition M t*) = *inputs M*
outputs (*add-transition M t*) = *outputs M*
transitions (*add-transition M t*) = *insert t* (*transitions M*)
states (*add-transition M t*) = *states M* ⟨*proof*⟩

lift-definition *add-state* :: ($'a,'b,'c$) *fsm* ⇒ $'a$ ⇒ ($'a,'b,'c$) *fsm* **is** *FSM-Impl.add-state*
⟨*proof*⟩

lemma *add-state-simps*[*simp*]:
initial (*add-state M q*) = *initial M*
inputs (*add-state M q*) = *inputs M*
outputs (*add-state M q*) = *outputs M*
transitions (*add-state M q*) = *transitions M*
states (*add-state M q*) = *insert q* (*states M*) ⟨*proof*⟩

lift-definition *add-input* :: ($'a,'b,'c$) *fsm* ⇒ $'b$ ⇒ ($'a,'b,'c$) *fsm* **is** *FSM-Impl.add-input*
⟨*proof*⟩

lemma *add-input-simps*[*simp*]:
initial (*add-input M x*) = *initial M*
inputs (*add-input M x*) = *insert x* (*inputs M*)
outputs (*add-input M x*) = *outputs M*
transitions (*add-input M x*) = *transitions M*
states (*add-input M x*) = *states M* ⟨*proof*⟩

lift-definition *add-output* :: ($'a,'b,'c$) *fsm* ⇒ $'c$ ⇒ ($'a,'b,'c$) *fsm* **is** *FSM-Impl.add-output*
⟨*proof*⟩

lemma *add-output-simps*[*simp*]:
initial (*add-output M y*) = *initial M*
inputs (*add-output M y*) = *inputs M*
outputs (*add-output M y*) = *insert y* (*outputs M*)
transitions (*add-output M y*) = *transitions M*
states (*add-output M y*) = *states M* ⟨*proof*⟩

lift-definition *add-transition-with-components* :: ('a,'b,'c) *fsm* ⇒ ('a,'b,'c) *transition* ⇒ ('a,'b,'c) *fsm* **is** *FSM-Impl.add-transition-with-components*
⟨*proof*⟩

lemma *add-transition-with-components-simps*[*simp*]:
initial (*add-transition-with-components M t*) = *initial M*
inputs (*add-transition-with-components M t*) = *insert* (*t-input t*) (*inputs M*)
outputs (*add-transition-with-components M t*) = *insert* (*t-output t*) (*outputs M*)
transitions (*add-transition-with-components M t*) = *insert t* (*transitions M*)
states (*add-transition-with-components M t*) = *insert* (*t-target t*) (*insert* (*t-source t*) (*states M*))
⟨*proof*⟩

4.26 Renaming Elements

lift-definition *rename-states* :: ('a,'b,'c) *fsm* ⇒ ('a ⇒ 'd) ⇒ ('d,'b,'c) *fsm* **is** *FSM-Impl.rename-states*
⟨*proof*⟩

lemma *rename-states-simps*[*simp*]:
initial (*rename-states M f*) = *f* (*initial M*)
states (*rename-states M f*) = *f* ‘ (*states M*)
inputs (*rename-states M f*) = *inputs M*
outputs (*rename-states M f*) = *outputs M*
transitions (*rename-states M f*) = (λ*t* . (*f* (*t-source t*), *t-input t*, *t-output t*, *f* (*t-target t*))) ‘ *transitions M*
⟨*proof*⟩

lemma *rename-states-isomorphism-language-state* :
assumes *bij-betw f* (*states M*) (*f* ‘ *states M*)
and *q* ∈ *states M*
shows *LS* (*rename-states M f*) (*f q*) = *LS M q*
⟨*proof*⟩

lemma *rename-states-isomorphism-language* :
assumes *bij-betw f* (*states M*) (*f* ‘ *states M*)
shows *L* (*rename-states M f*) = *L M*
⟨*proof*⟩

lemma *rename-states-observable* :
assumes *bij-betw f* (*states M*) (*f* ‘ *states M*)
and *observable M*
shows *observable* (*rename-states M f*)
⟨*proof*⟩

lemma *rename-states-minimal* :
assumes *bij-betw f* (*states M*) (*f* ' *states M*)
and *minimal M*
shows *minimal* (*rename-states M f*)
⟨*proof*⟩

fun *index-states* :: ('*a*::*linorder*,'*b*,'*c*) *fsm* ⇒ (*nat*,'*b*,'*c*) *fsm* **where**
index-states M = *rename-states M* (*assign-indices* (*states M*))

lemma *assign-indices-bij-betw*: *bij-betw* (*assign-indices* (*FSM.states M*)) (*FSM.states M*) (*assign-indices* (*FSM.states M*) ' *FSM.states M*)
⟨*proof*⟩

lemma *index-states-language* :
L (*index-states M*) = *L M*
⟨*proof*⟩

lemma *index-states-observable* :
assumes *observable M*
shows *observable* (*index-states M*)
⟨*proof*⟩

lemma *index-states-minimal* :
assumes *minimal M*
shows *minimal* (*index-states M*)
⟨*proof*⟩

fun *index-states-integer* :: ('*a*::*linorder*,'*b*,'*c*) *fsm* ⇒ (*integer*,'*b*,'*c*) *fsm* **where**
index-states-integer M = *rename-states M* (*integer-of-nat* ∘ *assign-indices* (*states M*))

lemma *assign-indices-integer-bij-betw*: *bij-betw* (*integer-of-nat* ∘ *assign-indices* (*states M*)) (*FSM.states M*) ((*integer-of-nat* ∘ *assign-indices* (*states M*)) ' *FSM.states M*)
⟨*proof*⟩

lemma *index-states-integer-language* :
L (*index-states-integer M*) = *L M*
⟨*proof*⟩

lemma *index-states-integer-observable* :
assumes *observable M*
shows *observable* (*index-states-integer M*)
⟨*proof*⟩

lemma *index-states-integer-minimal* :
  assumes *minimal M*
  shows *minimal* (*index-states-integer M*)
  ⟨*proof*⟩

4.27 Canonical Separators

lift-definition *canonical-separator′* :: (′*a*,′*b*,′*c*) *fsm* ⇒ ((′*a* × ′*a*),′*b*,′*c*) *fsm* ⇒ ′*a* ⇒ ′*a* ⇒ ((′*a* × ′*a*) + ′*a*,′*b*,′*c*) *fsm* **is** *FSM-Impl.canonical-separator′*
⟨*proof*⟩

lemma *canonical-separator′-simps* :
  assumes *initial P* = (*q1*,*q2*)
  shows *initial* (*canonical-separator′ M P q1 q2*) = *Inl* (*q1*,*q2*)
    states (*canonical-separator′ M P q1 q2*) = (*image Inl* (*states P*)) ∪ {*Inr q1*, *Inr q2*}
    inputs (*canonical-separator′ M P q1 q2*) = *inputs M* ∪ *inputs P*
    outputs (*canonical-separator′ M P q1 q2*) = *outputs M* ∪ *outputs P*
    transitions (*canonical-separator′ M P q1 q2*)
      = *shifted-transitions* (*transitions P*)
        ∪ *distinguishing-transitions* (*h-out M*) *q1 q2* (*states P*) (*inputs P*)
  ⟨*proof*⟩

lemma *canonical-separator′-simps-without-assm* :
    initial (*canonical-separator′ M P q1 q2*) = *Inl* (*q1*,*q2*)
    states (*canonical-separator′ M P q1 q2*) = (*if initial P* = (*q1*,*q2*) *then* (*image Inl* (*states P*)) ∪ {*Inr q1*, *Inr q2*} *else* {*Inl* (*q1*,*q2*)})
    inputs (*canonical-separator′ M P q1 q2*) = (*if initial P* = (*q1*,*q2*) *then inputs M* ∪ *inputs P else* {})
    outputs (*canonical-separator′ M P q1 q2*) = (*if initial P* = (*q1*,*q2*) *then outputs M* ∪ *outputs P else* {})
    transitions (*canonical-separator′ M P q1 q2*) = (*if initial P* = (*q1*,*q2*) *then shifted-transitions* (*transitions P*) ∪ *distinguishing-transitions* (*h-out M*) *q1 q2* (*states P*) (*inputs P*) *else* {})
  ⟨*proof*⟩

end

5 Product Machines

This theory defines the construction of product machines. A product machine of two finite state machines essentially represents all possible parallel executions of those two machines.

theory *Product-FSM*
imports *FSM*
begin

lift-definition *product* :: $('a,'b,'c)$ *fsm* ⇒ $('d,'b,'c)$ *fsm* ⇒ $('a \times 'd,'b,'c)$ *fsm* **is** *FSM-Impl.product*
⟨*proof*⟩

abbreviation *left-path p* ≡ *map* (λ*t*. (*fst* (*t-source t*), *t-input t*, *t-output t*, *fst* (*t-target t*))) *p*
abbreviation *right-path p* ≡ *map* (λ*t*. (*snd* (*t-source t*), *t-input t*, *t-output t*, *snd* (*t-target t*))) *p*
abbreviation *zip-path p1 p2* ≡ (*map* (λ *t* . ((*t-source* (*fst t*), *t-source* (*snd t*)), *t-input* (*fst t*), *t-output* (*fst t*), (*t-target* (*fst t*), *t-target* (*snd t*))))
(*zip p1 p2*))

lemma *product-simps*[*simp*]:
initial (*product A B*) = (*initial A*, *initial B*)
states (*product A B*) = (*states A*) × (*states B*)
inputs (*product A B*) = *inputs A* ∪ *inputs B*
outputs (*product A B*) = *outputs A* ∪ *outputs B*
⟨*proof*⟩

lemma *product-transitions-def* :
transitions (*product A B*) = {((*qA*,*qB*),*x*,*y*,(*qA'*,*qB'*)) | *qA qB x y qA' qB'* . (*qA*,*x*,*y*,*qA'*) ∈ *transitions A* ∧ (*qB*,*x*,*y*,*qB'*) ∈ *transitions B*}
⟨*proof*⟩

lemma *product-transitions-alt-def* :
transitions (*product A B*) = {((*t-source tA*, *t-source tB*),*t-input tA*, *t-output tA*, (*t-target tA*, *t-target tB*)) | *tA tB* . *tA* ∈ *transitions A* ∧ *tB* ∈ *transitions B* ∧ *t-input tA* = *t-input tB* ∧ *t-output tA* = *t-output tB*}
(**is** *?T1* = *?T2*)
⟨*proof*⟩

lemma *zip-path-last* : *length xs* = *length ys* ⟹ (*zip-path* (*xs* @ [*x*]) (*ys* @ [*y*])) = (*zip-path xs ys*)@(*zip-path* [*x*] [*y*])
⟨*proof*⟩

lemma *product-path-from-paths* :
assumes *path A* (*initial A*) *p1*
and *path B* (*initial B*) *p2*
and *p-io p1* = *p-io p2*
shows *path* (*product A B*) (*initial* (*product A B*)) (*zip-path p1 p2*)
and *target* (*initial* (*product A B*)) (*zip-path p1 p2*) = (*target* (*initial A*) *p1*,

target (*initial B*) *p2*)
⟨*proof*⟩

lemma *paths-from-product-path* :
assumes *path* (*product A B*) (*initial* (*product A B*)) *p*
shows *path A* (*initial A*) (*left-path p*)
and *path B* (*initial B*) (*right-path p*)
and *target* (*initial A*) (*left-path p*) = *fst* (*target* (*initial* (*product A B*)) *p*)
and *target* (*initial B*) (*right-path p*) = *snd* (*target* (*initial* (*product A B*)) *p*)
⟨*proof*⟩

lemma *zip-path-left-right*[*simp*] :
(*zip-path* (*left-path p*) (*right-path p*)) = *p* ⟨*proof*⟩

lemma *product-reachable-state-paths* :
assumes (*q1*,*q2*) ∈ *reachable-states* (*product A B*)
obtains *p1 p2*
where *path A* (*initial A*) *p1*
and *path B* (*initial B*) *p2*
and *target* (*initial A*) *p1* = *q1*
and *target* (*initial B*) *p2* = *q2*
and *p-io p1* = *p-io p2*
and *path* (*product A B*) (*initial* (*product A B*)) (*zip-path p1 p2*)
and *target* (*initial* (*product A B*)) (*zip-path p1 p2*) = (*q1*,*q2*)
⟨*proof*⟩

lemma *product-reachable-states*[*iff*] :
(*q1*,*q2*) ∈ *reachable-states* (*product A B*) ⟷ (∃ *p1 p2* . *path A* (*initial A*) *p1* ∧ *path B* (*initial B*) *p2* ∧ *target* (*initial A*) *p1* = *q1* ∧ *target* (*initial B*) *p2* = *q2* ∧ *p-io p1* = *p-io p2*)
⟨*proof*⟩

lemma *left-path-zip* : *length p1* = *length p2* ⟹ *left-path* (*zip-path p1 p2*) = *p1*
⟨*proof*⟩

lemma *right-path-zip* : *length p1* = *length p2* ⟹ *p-io p1* = *p-io p2* ⟹ *right-path* (*zip-path p1 p2*) = *p2*
⟨*proof*⟩

lemma *zip-path-append-left-right* : *length p1* = *length p2* ⟹ *zip-path* (*p1*@(*left-path p*)) (*p2*@(*right-path p*)) = (*zip-path p1 p2*)@*p*
⟨*proof*⟩

lemma *product-path*:
path (*product A B*) (*q1*,*q2*) *p* ⟷ (*path A q1* (*left-path p*) ∧ *path B q2* (*right-path p*))
⟨*proof*⟩

lemma *product-path-rev*:
assumes *p-io p1* = *p-io p2*
shows *path* (*product A B*) (*q1*,*q2*) (*zip-path p1 p2*) ⟷ (*path A q1 p1* ∧ *path B q2 p2*)
⟨*proof*⟩

lemma *product-language-state* :
shows *LS* (*product A B*) (*q1*,*q2*) = *LS A q1* ∩ *LS B q2*
⟨*proof*⟩

lemma *product-language* : *L* (*product A B*) = *L A* ∩ *L B*
⟨*proof*⟩

lemma *product-transition-split-ob* :
assumes *t* ∈ *transitions* (*product A B*)
obtains *t1 t2*
where *t1* ∈ *transitions A* ∧ *t-source t1* = *fst* (*t-source t*) ∧ *t-input t1* = *t-input t* ∧ *t-output t1* = *t-output t* ∧ *t-target t1* = *fst* (*t-target t*)
and *t2* ∈ *transitions B* ∧ *t-source t2* = *snd* (*t-source t*) ∧ *t-input t2* = *t-input t* ∧ *t-output t2* = *t-output t* ∧ *t-target t2* = *snd* (*t-target t*)
⟨*proof*⟩

lemma *product-transition-split* :
assumes *t* ∈ *transitions* (*product A B*)
shows (*fst* (*t-source t*), *t-input t*, *t-output t*, *fst* (*t-target t*)) ∈ *transitions A*
and (*snd* (*t-source t*), *t-input t*, *t-output t*, *snd* (*t-target t*)) ∈ *transitions B*
⟨*proof*⟩

lemma *product-target-split*:
assumes *target* (*q1*,*q2*) *p* = (*q1'*,*q2'*)
shows *target q1* (*left-path p*) = *q1'*
and *target q2* (*right-path p*) = *q2'*
⟨*proof*⟩

lemma *target-single-transition*[*simp*] : *target q1* [(*q1*, *x*, *y*, *q1'*)] = *q1'*

⟨*proof*⟩

lemma *product-undefined-input* :
assumes ¬ (∃ *t* ∈ *transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*)).
t-source t = *qq* ∧ *t-input t* = *x*)
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows ¬ (∃ *t1* ∈ *transitions M*. ∃ *t2* ∈ *transitions M*.
t-source t1 = *fst qq* ∧
t-source t2 = *snd qq* ∧
t-input t1 = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2*)
⟨*proof*⟩

5.1 Product Machines and Changing Initial States

lemma *product-from-reachable-next* :
assumes ((*q1*,*q2*),*x*,*y*,(*q1'*,*q2'*)) ∈ *transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows (*from-FSM* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*q1'*, *q2'*)) = (*product* (*from-FSM M q1'*) (*from-FSM M q2'*))
(**is** *?P1* = *?P2*)
⟨*proof*⟩

lemma *from-FSM-product-inputs* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows (*inputs* (*product* (*from-FSM M q1*) (*from-FSM M q2*))) = (*inputs M*)
⟨*proof*⟩

lemma *from-FSM-product-outputs* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows (*outputs* (*product* (*from-FSM M q1*) (*from-FSM M q2*))) = (*outputs M*)
⟨*proof*⟩

lemma *from-FSM-product-initial* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows *initial* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) = (*q1*,*q2*)
⟨*proof*⟩

lemma *product-from-reachable-next'* :
assumes *t* ∈ *transitions* (*product* (*from-FSM M* (*fst* (*t-source t*))) (*from-FSM M* (*snd* (*t-source t*))))
and *fst* (*t-source t*) ∈ *states M*

and *snd* (*t-source t*) ∈ *states M*
shows (*from-FSM* (*product* (*from-FSM M* (*fst* (*t-source t*))) (*from-FSM M* (*snd* (*t-source t*)))) (*fst* (*t-target t*),*snd* (*t-target t*))) = (*product* (*from-FSM M* (*fst* (*t-target t*))) (*from-FSM M* (*snd* (*t-target t*))))
⟨*proof*⟩

lemma *product-from-reachable-next′-path* :
assumes *t* ∈ *transitions* (*product* (*from-FSM M* (*fst* (*t-source t*))) (*from-FSM M* (*snd* (*t-source t*))))
and *fst* (*t-source t*) ∈ *states M*
and *snd* (*t-source t*) ∈ *states M*
shows *path* (*from-FSM* (*product* (*from-FSM M* (*fst* (*t-source t*))) (*from-FSM M* (*snd* (*t-source t*)))) (*fst* (*t-target t*),*snd* (*t-target t*))) (*fst* (*t-target t*),*snd* (*t-target t*)) *p* = *path* (*product* (*from-FSM M* (*fst* (*t-target t*))) (*from-FSM M* (*snd* (*t-target t*)))) (*fst* (*t-target t*),*snd* (*t-target t*)) *p*
(**is** *path ?P1 ?q p* = *path ?P2 ?q p*)
⟨*proof*⟩

lemma *product-from-transition*:
assumes (*q1′*,*q2′*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *transitions* (*product* (*from-FSM M q1′*) (*from-FSM M q2′*)) = *transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
⟨*proof*⟩

lemma *product-from-path*:
assumes (*q1′*,*q2′*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *path* (*product* (*from-FSM M q1′*) (*from-FSM M q2′*)) (*q1′*,*q2′*) *p*
shows *path* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*q1′*,*q2′*) *p*
⟨*proof*⟩

lemma *product-from-path-previous* :
assumes *path* (*product* (*from-FSM M* (*fst* (*t-target t*)))
(*from-FSM M* (*snd* (*t-target t*))))
(*t-target t*) *p* (**is** *path ?Pt* (*t-target t*) *p*)
and *t* ∈ *transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *path* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*t-target t*) *p* (**is** *path ?P* (*t-target t*) *p*)
⟨*proof*⟩

lemma *product-from-transition-shared-state* :
assumes *t* ∈ *transitions* (*product* (*from-FSM M q1′*) (*from-FSM M q2′*))
and (*q1′*,*q2′*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *t* ∈ *transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
⟨*proof*⟩

lemma *product-from-not-completely-specified* :
assumes ¬ *completely-specified-state* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*q1′*,*q2′*)
and (*q1′*,*q2′*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows ¬ *completely-specified-state* (*product* (*from-FSM M q1′*) (*from-FSM M q2′*)) (*q1′*,*q2′*)
⟨*proof*⟩

lemma *from-product-initial-paths-ex* :
assumes *q1* ∈ *states M*
and *q2* ∈ *states M*
shows (∃ *p1 p2*.
path (*from-FSM M q1*) (*initial* (*from-FSM M q1*)) *p1* ∧
path (*from-FSM M q2*) (*initial* (*from-FSM M q2*)) *p2* ∧
target (*initial* (*from-FSM M q1*)) *p1* = *q1* ∧
target (*initial* (*from-FSM M q2*)) *p2* = *q2* ∧ *p-io p1* = *p-io p2*)
⟨*proof*⟩

lemma *product-observable* :
assumes *observable M1*
and *observable M2*
shows *observable* (*product M1 M2*) (**is** *observable ?P*)
⟨*proof*⟩

lemma *product-observable-self-transitions* :
assumes *q* ∈ *reachable-states* (*product M M*)
and *observable M*
shows *fst q* = *snd q*
⟨*proof*⟩

lemma *zip-path-eq-left* :
assumes *length xs1* = *length xs2*
and *length xs2* = *length ys1*

and *length ys1 = length ys2*
and *zip-path xs1 xs2 = zip-path ys1 ys2*
shows *xs1 = ys1*
⟨*proof*⟩

lemma *zip-path-eq-right* :
assumes *length xs1 = length xs2*
and *length xs2 = length ys1*
and *length ys1 = length ys2*
and *p-io xs2 = p-io ys2*
and *zip-path xs1 xs2 = zip-path ys1 ys2*
shows *xs2 = ys2*
⟨*proof*⟩

lemma *zip-path-merge* :
(*zip-path* (*left-path p*) (*right-path p*)) = *p*
⟨*proof*⟩

lemma *product-from-reachable-path'* :
assumes *path* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*q1'*, *q2'*) *p*
and *q1* ∈ *reachable-states M*
and *q2* ∈ *reachable-states M*
shows *path* (*product* (*from-FSM M q1'*) (*from-FSM M q2'*)) (*q1'*, *q2'*) *p*
⟨*proof*⟩

lemma *product-from* :
assumes *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *product* (*from-FSM M q1*) (*from-FSM M q2*) = *from-FSM* (*product M M*) (*q1*,*q2*) (**is** *?PF = ?FP*)
⟨*proof*⟩

lemma *product-from-from* :
assumes (*q1'*,*q2'*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows (*product* (*from-FSM M q1'*) (*from-FSM M q2'*)) = (*from-FSM* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*q1'*,*q2'*))
⟨*proof*⟩

lemma *submachine-transition-product-from* :
assumes *is-submachine S* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
and ((*q1*,*q2*),*x*,*y*,(*q1'*,*q2'*)) ∈ *transitions S*

and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *is-submachine* (*from-FSM S* (*q1′*,*q2′*)) (*product* (*from-FSM M q1′*) (*from-FSM M q2′*))
⟨*proof*⟩

lemma *submachine-transition-complete-product-from* :
assumes *is-submachine S* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
and *completely-specified S*
and ((*q1*,*q2*),*x*,*y*,(*q1′*,*q2′*)) ∈ *transitions S*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *completely-specified* (*from-FSM S* (*q1′*,*q2′*))
⟨*proof*⟩

5.2 Calculating Acyclic Intersection Languages

lemma *acyclic-product* :
assumes *acyclic B*
shows *acyclic* (*product A B*)
⟨*proof*⟩

lemma *acyclic-product-path-length* :
assumes *acyclic B*
and *path* (*product A B*) (*initial* (*product A B*)) *p*
shows *length p* < *size B*
⟨*proof*⟩

lemma *acyclic-language-alt-def* :
assumes *acyclic A*
shows *image p-io* (*acyclic-paths-up-to-length A* (*initial A*) (*size A* − *1*)) = *L A*
⟨*proof*⟩

definition *acyclic-language-intersection* :: (′*a*,′*b*,′*c*) *fsm* ⇒ (′*d*,′*b*,′*c*) *fsm* ⇒ (′*b* × ′*c*) *list set* **where**
acyclic-language-intersection M A = (*let P* = *product M A in image p-io* (*acyclic-paths-up-to-length P* (*initial P*) (*size A* − *1*)))

lemma *acyclic-language-intersection-completeness* :
assumes *acyclic A*
shows *acyclic-language-intersection M A* = *L M* ∩ *L A*
⟨*proof*⟩

end

6 Minimisation by OFSM Tables

This theory presents the classical algorithm for transforming observable FSMs into language-equivalent observable and minimal FSMs in analogy to the minimisation of finite automata.

theory *Minimisation*
imports *FSM*
begin

6.1 OFSM Tables

OFSM tables partition the states of an FSM based on an initial partition and an iteration counter. States are in the same element of the 0th table iff they are in the same element of the initial partition. States q1, q2 are in the same element of the (k+1)-th table if they are in the same element of the k-th table and furthermore for each IO pair (x,y) either (x,y) is not in the language of both q1 and q2 or it is in the language of both states and the states q1', q2' reached via (x,y) from q1 and q2, respectively, are in the same element of the k-th table.

fun *ofsm-table* :: ('a,'b,'c) *fsm* ⇒ ('a ⇒ 'a *set*) ⇒ *nat* ⇒ 'a ⇒ 'a *set* **where**
  ofsm-table M f 0 q = (*if q* ∈ *states M then f q else* {}) |
  ofsm-table M f (*Suc k*) *q* = (*let*
    prev-table = *ofsm-table M f k*
   in {*q'* ∈ *prev-table q* . ∀ *x* ∈ *inputs M* . ∀ *y* ∈ *outputs M* . (*case h-obs M q x y of Some qT* ⇒ (*case h-obs M q' x y of Some qT'* ⇒ *prev-table qT* = *prev-table qT'* | *None* ⇒ *False*) | *None* ⇒ *h-obs M q' x y* = *None*) })

lemma *ofsm-table-non-state* :
  assumes *q* ∉ *states M*
  shows *ofsm-table M f k q* = {}
⟨*proof*⟩

lemma *ofsm-table-subset*:
  assumes *i* ≤ *j*
  shows *ofsm-table M f j q* ⊆ *ofsm-table M f i q*
⟨*proof*⟩

lemma *ofsm-table-case-helper* :
  (*case h-obs M q x y of Some qT* ⇒ (*case h-obs M q' x y of Some qT'* ⇒ *ofsm-table M f k qT* = *ofsm-table M f k qT'* | *None* ⇒ *False*) | *None* ⇒ *h-obs M q' x y* = *None*)
   = ((∃ *qT qT'* . *h-obs M q x y* = *Some qT* ∧ *h-obs M q' x y* = *Some qT'* ∧ *ofsm-table M f k qT* = *ofsm-table M f k qT'*) ∨ (*h-obs M q x y* = *None* ∧ *h-obs M q' x y* = *None*))
⟨*proof*⟩

lemma *ofsm-table-case-helper-neg* :
  $(\neg$ (*case h-obs M q x y of Some qT* $\Rightarrow$ (*case h-obs M q' x y of Some qT'* $\Rightarrow$ *ofsm-table M f k qT* $=$ *ofsm-table M f k qT'* | *None* $\Rightarrow$ *False*) | *None* $\Rightarrow$ *h-obs M q' x y* $=$ *None*))
  $=$ (($\exists$ *qT qT'* . *h-obs M q x y* $=$ *Some qT* $\wedge$ *h-obs M q' x y* $=$ *Some qT'* $\wedge$ *ofsm-table M f k qT* $\neq$ *ofsm-table M f k qT'*) $\vee$ (*h-obs M q x y* $=$ *None* $\longleftrightarrow$ *h-obs M q' x y* $\neq$ *None*))
  ⟨*proof*⟩

lemma *ofsm-table-fixpoint* :
  assumes $i \leq j$
  and $\bigwedge q . q \in$ *states M* $\Longrightarrow$ *ofsm-table M f* (*Suc i*) *q* $=$ *ofsm-table M f i q*
  and $q \in$ *states M*
shows *ofsm-table M f j q* $=$ *ofsm-table M f i q*
⟨*proof*⟩

function *ofsm-table-fix* :: ('*a*,'*b*,'*c*) *fsm* $\Rightarrow$ ('*a* $\Rightarrow$ '*a set*) $\Rightarrow$ *nat* $\Rightarrow$ '*a* $\Rightarrow$ '*a set*
where
  ofsm-table-fix M f k $=$ (*let*
    cur-table $=$ *ofsm-table M* (λq. *f q* $\cap$ *states M*) *k*;
    next-table $=$ *ofsm-table M* (λq. *f q* $\cap$ *states M*) (*Suc k*)
  in if ($\forall$ $q \in$ *states M* . *cur-table q* $=$ *next-table q*)
    then cur-table
    else ofsm-table-fix M f (*Suc k*))
  ⟨*proof*⟩
termination
⟨*proof*⟩

lemma *ofsm-table-restriction-to-states* :
  assumes $\bigwedge q . q \in$ *states M* $\Longrightarrow$ *f q* $\subseteq$ *states M*
  and $q \in$ *states M*
shows *ofsm-table M f k q* $=$ *ofsm-table M* (λq . *f q* $\cap$ *states M*) *k q*
⟨*proof*⟩

lemma *ofsm-table-fix-length* :
  assumes $\bigwedge q . q \in$ *states M* $\Longrightarrow$ *f q* $\subseteq$ *states M*
  obtains *k* **where** $\bigwedge q . q \in$ *states M* $\Longrightarrow$ *ofsm-table-fix M f 0 q* $=$ *ofsm-table M f k q* **and** $\bigwedge q\ k' . q \in$ *states M* $\Longrightarrow k' \geq k \Longrightarrow$ *ofsm-table M f k' q* $=$ *ofsm-table M f k q*
⟨*proof*⟩

lemma *ofsm-table-containment* :
  assumes $q \in$ *states M*
  and $\bigwedge q .\ q \in$ *states M* $\Longrightarrow q \in f\ q$
  shows $q \in$ *ofsm-table M f k q*
⟨*proof*⟩

lemma *ofsm-table-states* :
  assumes $\bigwedge q .\ q \in$ *states M* $\Longrightarrow f\ q \subseteq$ *states M*
  and $q \in$ *states M*
shows *ofsm-table M f k q* $\subseteq$ *states M*
⟨*proof*⟩

6.1.1 Properties of Initial Partitions

definition *equivalence-relation-on-states* :: ('a,'b,'c) *fsm* ⇒ ('a ⇒ 'a *set*) ⇒ *bool*
where
  equivalence-relation-on-states M f =
    (*equiv* (*states M*) $\{(q1,q2) \mid q1\ q2 .\ q1 \in$ *states M* $\wedge\ q2 \in f\ q1\}$
    $\wedge$ ($\forall\ q \in$ *states M* . $f\ q \subseteq$ *states M*))

lemma *equivalence-relation-on-states-refl* :
  assumes *equivalence-relation-on-states M f*
  and $q \in$ *states M*
shows $q \in f\ q$
  ⟨*proof*⟩

lemma *equivalence-relation-on-states-sym* :
  assumes *equivalence-relation-on-states M f*
  and $q1 \in$ *states M*
  and $q2 \in f\ q1$
shows $q1 \in f\ q2$
  ⟨*proof*⟩

lemma *equivalence-relation-on-states-trans* :
  assumes *equivalence-relation-on-states M f*
  and $q1 \in$ *states M*
  and $q2 \in f\ q1$
  and $q3 \in f\ q2$
shows $q3 \in f\ q1$
⟨*proof*⟩

lemma *equivalence-relation-on-states-ran* :
  assumes *equivalence-relation-on-states M f*
  and $q \in$ *states M*
shows $f\ q \subseteq$ *states M*
  ⟨*proof*⟩

6.1.2 Properties of OFSM tables for initial partitions based on equivalence relations

lemma *h-obs-io* :
assumes *h-obs M q x y = Some q'*
shows $x \in$ *inputs M* **and** $y \in$ *outputs M*
⟨*proof*⟩

lemma *ofsm-table-language* :
assumes $q' \in$ *ofsm-table M f k q*
and *length io* $\leq k$
and $q \in$ *states M*
and *equivalence-relation-on-states M f*
shows *is-in-language M q io* $\longleftrightarrow$ *is-in-language M q' io*
and *is-in-language M q io* $\Longrightarrow$ (*after M q' io*) $\in f$ (*after M q io*)
⟨*proof*⟩

lemma *after-is-state-is-in-language* :
assumes $q \in$ *states M*
and *is-in-language M q io*
shows *FSM.after M q io* $\in$ *states M*
⟨*proof*⟩

lemma *ofsm-table-elem* :
assumes $q \in$ *states M*
and $q' \in$ *states M*
and *equivalence-relation-on-states M f*
and $\bigwedge$ *io . length io* $\leq k \Longrightarrow$ *is-in-language M q io* $\longleftrightarrow$ *is-in-language M q' io*
and $\bigwedge$ *io . length io* $\leq k \Longrightarrow$ *is-in-language M q io* $\Longrightarrow$ (*after M q' io*) $\in f$ (*after M q io*)
shows $q' \in$ *ofsm-table M f k q*
⟨*proof*⟩

lemma *ofsm-table-set* :
assumes $q \in$ *states M*
and *equivalence-relation-on-states M f*
shows *ofsm-table M f k q* = {*q' . q'* $\in$ *states M* $\wedge$ ($\forall$ *io . length io* $\leq k \longrightarrow$ (*is-in-language M q io* $\longleftrightarrow$ *is-in-language M q' io*) $\wedge$ (*is-in-language M q io* $\longrightarrow$ *after M q' io* $\in f$ (*after M q io*)))}
⟨*proof*⟩

lemma *ofsm-table-set-observable* :
assumes *observable M* **and** $q \in$ *states M*
and *equivalence-relation-on-states M f*

shows *ofsm-table M f k q* = {*q′* . *q′* ∈ *states M* ∧ (∀ *io* . *length io* ≤ *k* ⟶ (*io* ∈ *LS M q* ⟷ *io* ∈ *LS M q′*) ∧ (*io* ∈ *LS M q* ⟶ *after M q′ io* ∈ *f* (*after M q io*)))}
⟨*proof*⟩

lemma *ofsm-table-eq-if-elem* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M* **and** *equivalence-relation-on-states M f*
shows (*ofsm-table M f k q1* = *ofsm-table M f k q2*) = (*q2* ∈ *ofsm-table M f k q1*)
⟨*proof*⟩

lemma *ofsm-table-fix-language* :
fixes *M* :: (′*a*,′*b*,′*c*) *fsm*
assumes *q′* ∈ *ofsm-table-fix M f 0 q*
and *q* ∈ *states M*
and *observable M*
and *equivalence-relation-on-states M f*
shows *LS M q* = *LS M q′*
and *io* ∈ *LS M q* ⟹ *after M q′ io* ∈ *f* (*after M q io*)
⟨*proof*⟩

lemma *ofsm-table-same-language* :
assumes *LS M q* = *LS M q′*
and ⋀ *io* . *io* ∈ *LS M q* ⟹ *after M q′ io* ∈ *f* (*after M q io*)
and *observable M*
and *q′* ∈ *states M*
and *q* ∈ *states M*
and *equivalence-relation-on-states M f*
shows *ofsm-table M f k q* = *ofsm-table M f k q′*
⟨*proof*⟩

lemma *ofsm-table-fix-set* :
assumes *q* ∈ *states M*
and *observable M*
and *equivalence-relation-on-states M f*
shows *ofsm-table-fix M f 0 q* = {*q′* ∈ *states M* . *LS M q′* = *LS M q* ∧ (∀ *io* ∈ *LS M q* . *after M q′ io* ∈ *f* (*after M q io*))}
⟨*proof*⟩

lemma *ofsm-table-fix-eq-if-elem* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*

and *equivalence-relation-on-states M f*
shows (*ofsm-table-fix M f 0 q1* = *ofsm-table-fix M f 0 q2*) = (*q2* ∈ *ofsm-table-fix M f 0 q1*)
⟨*proof*⟩

lemma *ofsm-table-refinement-disjoint* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*
and *equivalence-relation-on-states M f*
and *ofsm-table M f k q1* ≠ *ofsm-table M f k q2*
shows *ofsm-table M f* (*Suc k*) *q1* ≠ *ofsm-table M f* (*Suc k*) *q2*
⟨*proof*⟩

lemma *ofsm-table-partition-finite* :
assumes *equivalence-relation-on-states M f*
shows *finite* (*ofsm-table M f k ‘ states M*)
⟨*proof*⟩

lemma *ofsm-table-refinement-card* :
assumes *equivalence-relation-on-states M f*
and *A* ⊆ *states M*
and *i* ≤ *j*
shows *card* (*ofsm-table M f j ‘ A*) ≥ *card* (*ofsm-table M f i ‘ A*)
⟨*proof*⟩

lemma *ofsm-table-refinement-card-fix-Suc* :
assumes *equivalence-relation-on-states M f*
and *card* (*ofsm-table M f* (*Suc k*) *‘ states M*) = *card* (*ofsm-table M f k ‘ states M*)
and *q* ∈ *states M*
shows *ofsm-table M f* (*Suc k*) *q* = *ofsm-table M f k q*
⟨*proof*⟩

lemma *ofsm-table-refinement-card-fix* :
assumes *equivalence-relation-on-states M f*
and *card* (*ofsm-table M f j ‘ states M*) = *card* (*ofsm-table M f i ‘ states M*)
and *q* ∈ *states M*
and *i* ≤ *j*
shows *ofsm-table M f j q* = *ofsm-table M f i q*
⟨*proof*⟩

lemma *ofsm-table-partition-fixpoint-Suc* :

assumes *equivalence-relation-on-states M f*
and $q \in$ *states M*
shows *ofsm-table M f* (*size M* $-$ *card* (*f* ‘ *states M*)) *q* = *ofsm-table M f* (*Suc* (*size M* $-$ *card* (*f* ‘ *states M*))) *q*
⟨*proof*⟩

lemma *ofsm-table-partition-fixpoint* :
assumes *equivalence-relation-on-states M f*
and *size M* $\leq m$
and $q \in$ *states M*
shows *ofsm-table M f* (m $-$ *card* (*f* ‘ *states M*)) *q* = *ofsm-table M f* (*Suc* (m $-$ *card* (*f* ‘ *states M*))) *q*
⟨*proof*⟩

lemma *ofsm-table-fix-partition-fixpoint* :
assumes *equivalence-relation-on-states M f*
and *size M* $\leq m$
and $q \in$ *states M*
shows *ofsm-table M f* (m $-$ *card* (*f* ‘ *states M*)) *q* = *ofsm-table-fix M f 0 q*
⟨*proof*⟩

6.2 A minimisation function based on OFSM-tables

lemma *language-equivalence-classes-preserve-observability*:
assumes *transitions M'* = (λ *t* . ({$q \in$ *states M* . *LS M q* = *LS M* (*t-source t*)} , *t-input t*, *t-output t*, {$q \in$ *states M* . *LS M q* = *LS M* (*t-target t*)})) ‘ *transitions M*
and *observable M*
shows *observable M'*
⟨*proof*⟩

lemma *language-equivalence-classes-retain-language-and-induce-minimality* :
assumes *transitions M'* = (λ *t* . ({$q \in$ *states M* . *LS M q* = *LS M* (*t-source t*)} , *t-input t*, *t-output t*, {$q \in$ *states M* . *LS M q* = *LS M* (*t-target t*)})) ‘ *transitions M*
and *states M'* = (λq . {$q' \in$ *states M* . *LS M q* = *LS M q'*}) ‘ *states M*
and *initial M'* = {$q' \in$ *states M* . *LS M q'* = *LS M* (*initial M*)}
and *observable M*
shows *L M* = *L M'*
and *minimal M'*
⟨*proof*⟩

fun *minimise* :: ('a :: linorder,'b :: linorder,'c :: linorder) fsm ⇒ ('a set,'b,'c) fsm
where
minimise M = (*let*
eq-class = *ofsm-table-fix M* (λ*q* . *states M*) *0*;
ts = (λ *t* . (*eq-class* (*t-source t*), *t-input t*, *t-output t*, *eq-class* (*t-target t*))) ' (*transitions M*);
q0 = *eq-class* (*initial M*);
eq-states = *eq-class* |'| *fstates M*;
M' = *create-unconnected-fsm-from-fsets q0 eq-states* (*finputs M*) (*foutputs M*)
in add-transitions M' ts)

lemma *minimise-initial-partition* :
equivalence-relation-on-states M (λ*q* . *states M*)
⟨*proof*⟩

lemma *minimise-props*:
assumes *observable M*
shows *initial* (*minimise M*) = {*q'* ∈ *states M* . *LS M q'* = *LS M* (*initial M*)}
and *states* (*minimise M*) = (λ*q* . {*q'* ∈ *states M* . *LS M q* = *LS M q'*}) ' *states M*
and *inputs* (*minimise M*) = *inputs M*
and *outputs* (*minimise M*) = *outputs M*
and *transitions* (*minimise M*) = (λ *t* . ({*q* ∈ *states M* . *LS M q* = *LS M* (*t-source t*)} , *t-input t*, *t-output t*, {*q* ∈ *states M* . *LS M q* = *LS M* (*t-target t*)})) ' *transitions M*
⟨*proof*⟩

lemma *minimise-observable*:
assumes *observable M*
shows *observable* (*minimise M*)
⟨*proof*⟩

lemma *minimise-minimal*:
assumes *observable M*
shows *minimal* (*minimise M*)
⟨*proof*⟩

lemma *minimise-language*:
assumes *observable M*
shows *L* (*minimise M*) = *L M*
⟨*proof*⟩

lemma *minimal-observable-code* :
assumes *observable M*

shows *minimal M* = (∀ *q* ∈ *states M* . *ofsm-table-fix M* (λ*q* . *states M*) *0 q* = {*q*})
⟨*proof*⟩

lemma *minimise-states-subset* :
assumes *observable M*
and *q* ∈ *states* (*minimise M*)
shows *q* ⊆ *states M*
⟨*proof*⟩

lemma *minimise-states-finite* :
assumes *observable M*
and *q* ∈ *states* (*minimise M*)
shows *finite q*
⟨*proof*⟩

end

7 Computation of distinguishing traces based on OFSM tables

This theory implements an algorithm for finding minimal length distinguishing traces for observable minimal FSMs based on OFSM tables.

theory *Distinguishability*
imports *Minimisation HOL.List*
begin

7.1 Finding Diverging OFSM Tables

definition *ofsm-table-fixpoint-value* :: (′*a*,′*b*,′*c*) *fsm* ⇒ *nat* **where**
ofsm-table-fixpoint-value M = (*SOME k* . (∀ *q* . *q* ∈ *states M* ⟶ *ofsm-table-fix M* (λ*q* . *states M*) *0 q* = *ofsm-table M* (λ*q* . *states M*) *k q*) ∧ (∀ *q k*′ . *q* ∈ *states M* ⟶ *k*′ ≥ *k* ⟶ *ofsm-table M* (λ*q* . *states M*) *k*′ *q* = *ofsm-table M* (λ*q* . *states M*) *k q*))

function *find-first-distinct-ofsm-table-gt* :: (′*a*,′*b*,′*c*) *fsm* ⇒ ′*a* ⇒ ′*a* ⇒ *nat* ⇒ *nat* **where**
find-first-distinct-ofsm-table-gt M q1 q2 k =
(*if q1* ∈ *states M* ∧ *q2* ∈ *states M* ∧ ((*ofsm-table-fix M* (λ*q* . *states M*) *0 q1* ≠ *ofsm-table-fix M* (λ*q* . *states M*) *0 q2*))
then (*if ofsm-table M* (λ*q* . *states M*) *k q1* ≠ *ofsm-table M* (λ*q* . *states M*) *k q2*
then k
else find-first-distinct-ofsm-table-gt M q1 q2 (*Suc k*))
else 0)
⟨*proof*⟩
termination

⟨*proof*⟩

partial-function (*tailrec*) *find-first-distinct-ofsm-table-no-check* :: ('*a*,'*b*,'*c*) *fsm* ⇒ '*a* ⇒ '*a* ⇒ *nat* ⇒ *nat* **where**
find-first-distinct-ofsm-table-no-check-def[*code*]:
find-first-distinct-ofsm-table-no-check M q1 q2 k =
(*if ofsm-table M* (λ*q* . *states M*) *k q1* ≠ *ofsm-table M* (λ*q* . *states M*) *k q2*
then k
else find-first-distinct-ofsm-table-no-check M q1 q2 (*Suc k*))

fun *find-first-distinct-ofsm-table-gt'* :: ('*a*,'*b*,'*c*) *fsm* ⇒ '*a* ⇒ '*a* ⇒ *nat* ⇒ *nat* **where**
find-first-distinct-ofsm-table-gt' M q1 q2 k =
(*if q1* ∈ *states M* ∧ *q2* ∈ *states M* ∧ ((*q2* ∉ *ofsm-table-fix M* (λ*q* . *states M*) *0 q1*))
then find-first-distinct-ofsm-table-no-check M q1 q2 k
else 0)

lemma *find-first-distinct-ofsm-table-gt-code*[*code*] :
find-first-distinct-ofsm-table-gt M q1 q2 k = *find-first-distinct-ofsm-table-gt' M q1 q2 k*
⟨*proof*⟩

lemma *find-first-distinct-ofsm-table-gt-is-first-gt* :
assumes *q1* ∈ *FSM.states M*
and *q2* ∈ *FSM.states M*
and *ofsm-table-fix M* (λ*q* . *states M*) *0 q1* ≠ *ofsm-table-fix M* (λ*q* . *states M*) *0 q2*
shows *ofsm-table M* (λ*q* . *states M*) (*find-first-distinct-ofsm-table-gt M q1 q2 k*) *q1* ≠ *ofsm-table M* (λ*q* . *states M*) (*find-first-distinct-ofsm-table-gt M q1 q2 k*) *q2*
and *k* ≤ *k'* ⟹ *k'* < (*find-first-distinct-ofsm-table-gt M q1 q2 k*) ⟹ *ofsm-table M* (λ*q* . *states M*) *k' q1* = *ofsm-table M* (λ*q* . *states M*) *k' q2*
⟨*proof*⟩

abbreviation(*input*) *find-first-distinct-ofsm-table M q1 q2* ≡ *find-first-distinct-ofsm-table-gt M q1 q2 0*

lemma *find-first-distinct-ofsm-table-is-first* :
assumes *q1* ∈ *FSM.states M*
and *q2* ∈ *FSM.states M*
and *ofsm-table-fix M* (λ*q* . *states M*) *0 q1* ≠ *ofsm-table-fix M* (λ*q* . *states M*) *0 q2*
shows *ofsm-table M* (λ*q* . *states M*) (*find-first-distinct-ofsm-table M q1 q2*) *q1* ≠ *ofsm-table M* (λ*q* . *states M*) (*find-first-distinct-ofsm-table M q1 q2*) *q2*
and *k'* < (*find-first-distinct-ofsm-table M q1 q2*) ⟹ *ofsm-table M* (λ*q* . *states M*) *k' q1* = *ofsm-table M* (λ*q* . *states M*) *k' q2*
⟨*proof*⟩

fun *select-diverging-ofsm-table-io* :: ('a,'b::linorder,'c::linorder) fsm ⇒ 'a ⇒ 'a ⇒ nat ⇒ ('b × 'c) × ('a option × 'a option) **where**

```
  select-diverging-ofsm-table-io M q1 q2 k = (let
      ins = inputs-as-list M;
      outs = outputs-as-list M;
      table = ofsm-table M (λq . states M) (k−1);
      f = (λ (x,y) . case (h-obs M q1 x y, h-obs M q2 x y)
                  of
                    (Some q1', Some q2')  ⇒ if table q1' ≠ table q2'
                                              then Some ((x,y),(Some q1', Some q2'))
                                              else None |
                    (None,None)           ⇒ None |
                    (Some q1', None)       ⇒ Some ((x,y),(Some q1', None)) |
                    (None, Some q2')       ⇒ Some ((x,y),(None, Some q2')))
      in
        hd (List.map-filter f (List.product ins outs)))
```

lemma *select-diverging-ofsm-table-io-Some* :
assumes *observable M*
and $q1 \in states\ M$
and $q2 \in states\ M$
and *ofsm-table M* (λq . *states M*) (*Suc k*) *q1* ≠ *ofsm-table M* (λq . *states M*) (*Suc k*) *q2*
obtains *x y*
where *select-diverging-ofsm-table-io M q1 q2* (*Suc k*) = ((*x*,*y*),(*h-obs M q1 x y*, *h-obs M q2 x y*))
and ⋀ *q1' q2'* . *h-obs M q1 x y* = *Some q1'* ⟹ *h-obs M q2 x y* = *Some q2'* ⟹ *ofsm-table M* (λq . *states M*) *k q1'* ≠ *ofsm-table M* (λq . *states M*) *k q2'*
and *h-obs M q1 x y* ≠ *None* ∨ *h-obs M q2 x y* ≠ *None*
⟨*proof*⟩

7.2 Assembling Distinguishing Traces

fun *assemble-distinguishing-sequence-from-ofsm-table* :: ('a,'b::linorder,'c::linorder) fsm ⇒ 'a ⇒ 'a ⇒ nat ⇒ ('b × 'c) list **where**

```
  assemble-distinguishing-sequence-from-ofsm-table M q1 q2 0 = [] |
  assemble-distinguishing-sequence-from-ofsm-table M q1 q2 (Suc k) = (case
      select-diverging-ofsm-table-io M q1 q2 (Suc k)
    of
    ((x,y),(Some q1',Some q2')) ⇒ (x,y) # (assemble-distinguishing-sequence-from-ofsm-table M q1' q2' k) |
      ((x,y),-)                  ⇒ [(x,y)])
```

lemma *assemble-distinguishing-sequence-from-ofsm-table-distinguishes* :

assumes *observable M*
and *q1 ∈ states M*
and *q2 ∈ states M*
and *ofsm-table M (λq . states M) k q1 ≠ ofsm-table M (λq . states M) k q2*
shows *assemble-distinguishing-sequence-from-ofsm-table M q1 q2 k ∈ LS M q1 ∪ LS M q2*
and *assemble-distinguishing-sequence-from-ofsm-table M q1 q2 k ∉ LS M q1 ∩ LS M q2*
and *butlast (assemble-distinguishing-sequence-from-ofsm-table M q1 q2 k) ∈ LS M q1 ∩ LS M q2*
⟨*proof*⟩

lemma *assemble-distinguishing-sequence-from-ofsm-table-length* :
length (assemble-distinguishing-sequence-from-ofsm-table M q1 q2 k) ≤ k
⟨*proof*⟩

lemma *ofsm-table-fix-partition-fixpoint-trivial-partition* :
assumes *q ∈ states M*
shows *ofsm-table-fix M (λq. FSM.states M) 0 q = ofsm-table M (λq. FSM.states M) (size M − 1) q*
⟨*proof*⟩

fun *get-distinguishing-sequence-from-ofsm-tables* :: (*'a,'b::linorder,'c::linorder*) *fsm* ⇒ *'a* ⇒ *'a* ⇒ (*'b* × *'c*) *list* **where**
get-distinguishing-sequence-from-ofsm-tables M q1 q2 = (let
k = find-first-distinct-ofsm-table M q1 q2
in assemble-distinguishing-sequence-from-ofsm-table M q1 q2 k)

lemma *get-distinguishing-sequence-from-ofsm-tables-is-distinguishing-trace* :
assumes *observable M*
and *minimal M*
and *q1 ∈ states M*
and *q2 ∈ states M*
and *q1 ≠ q2*
shows *get-distinguishing-sequence-from-ofsm-tables M q1 q2 ∈ LS M q1 ∪ LS M q2*
and *get-distinguishing-sequence-from-ofsm-tables M q1 q2 ∉ LS M q1 ∩ LS M q2*
and *butlast (get-distinguishing-sequence-from-ofsm-tables M q1 q2) ∈ LS M q1 ∩ LS M q2*
⟨*proof*⟩

lemma *get-distinguishing-sequence-from-ofsm-tables-distinguishes* :
assumes *observable M*
and *minimal M*

and *q1 ∈ states M*
and *q2 ∈ states M*
and *q1 ≠ q2*
shows *distinguishes M q1 q2 (get-distinguishing-sequence-from-ofsm-tables M q1 q2)*
⟨*proof*⟩

7.3 Minimal Distinguishing Traces

lemma *get-distinguishing-sequence-from-ofsm-tables-is-minimally-distinguishing* :
fixes *M* :: (*'a,'b::linorder,'c::linorder*) *fsm*
assumes *observable M*
and *minimal M*
and *q1 ∈ states M*
and *q2 ∈ states M*
and *q1 ≠ q2*
shows *minimally-distinguishes M q1 q2 (get-distinguishing-sequence-from-ofsm-tables M q1 q2)*
⟨*proof*⟩

lemma *minimally-distinguishes-length* :
assumes *observable M*
and *minimal M*
and *q1 ∈ states M*
and *q2 ∈ states M*
and *q1 ≠ q2*
and *minimally-distinguishes M q1 q2 io*
shows *length io ≤ size M − 1*
⟨*proof*⟩

end

8 Properties of Sets of IO Sequences

This theory contains various definitions for properties of sets of IO-traces.

theory *IO-Sequence-Set*
imports *FSM*
begin

fun *output-completion* :: (*'a × 'b*) *list set ⇒ 'b set ⇒* (*'a × 'b*) *list set* **where**
output-completion P Out = P ∪ {io@[(fst xy, y)] | io xy y . y ∈ Out ∧ io@[xy] ∈ P ∧ io@[(fst xy, y)] ∉ P}

fun *output-complete-sequences* :: (*'a,'b,'c*) *fsm ⇒* (*'b × 'c*) *list set ⇒ bool* **where**

output-complete-sequences M P = (∀ *io* ∈ *P* . *io* = [] ∨ (∀ *y* ∈ (*outputs M*) . (*butlast io*)@[(*fst* (*last io*), *y*)] ∈ *P*))

fun *acyclic-sequences* :: ('*a*,'*b*,'*c*) *fsm* ⇒ '*a* ⇒ ('*b* × '*c*) *list set* ⇒ *bool* **where**
acyclic-sequences M q P = (∀ *p* . (*path M q p* ∧ *p-io p* ∈ *P*) ⟶ *distinct* (*visited-states q p*))

fun *acyclic-sequences'* :: ('*a*,'*b*,'*c*) *fsm* ⇒ '*a* ⇒ ('*b* × '*c*) *list set* ⇒ *bool* **where**
acyclic-sequences' M q P = (∀ *io* ∈ *P* . ∀ *p* ∈ (*paths-for-io M q io*) . *distinct* (*visited-states q p*))

lemma *acyclic-sequences-alt-def*[*code*] : *acyclic-sequences M P* = *acyclic-sequences' M P*
⟨*proof*⟩

fun *single-input-sequences* :: ('*a*,'*b*,'*c*) *fsm* ⇒ ('*b* × '*c*) *list set* ⇒ *bool* **where**
single-input-sequences M P = (∀ *xys1 xys2 xy1 xy2* . (*xys1*@[*xy1*] ∈ *P* ∧ *xys2*@[*xy2*] ∈ *P* ∧ *io-targets M xys1* (*initial M*) = *io-targets M xys2* (*initial M*)) ⟶ *fst xy1* = *fst xy2*)

fun *single-input-sequences'* :: ('*a*,'*b*,'*c*) *fsm* ⇒ ('*b* × '*c*) *list set* ⇒ *bool* **where**
single-input-sequences' M P = (∀ *io1* ∈ *P* . ∀ *io2* ∈ *P* . *io1* = [] ∨ *io2* = [] ∨ ((*io-targets M* (*butlast io1*) (*initial M*) = *io-targets M* (*butlast io2*) (*initial M*)) ⟶ *fst* (*last io1*) = *fst* (*last io2*)))

lemma *single-input-sequences-alt-def*[*code*] : *single-input-sequences M P* = *single-input-sequences' M P*
⟨*proof*⟩

fun *output-complete-for-FSM-sequences-from-state* :: ('*a*,'*b*,'*c*) *fsm* ⇒ '*a* ⇒ ('*b* × '*c*) *list set* ⇒ *bool* **where**
output-complete-for-FSM-sequences-from-state M q P = (∀ *io xy t* . *io*@[*xy*] ∈ *P* ∧ *t* ∈ *transitions M* ∧ *t-source t* ∈ *io-targets M io q* ∧ *t-input t* = *fst xy* ⟶ *io*@[(*fst xy*, *t-output t*)] ∈ *P*)

lemma *output-complete-for-FSM-sequences-from-state-alt-def* :
shows *output-complete-for-FSM-sequences-from-state M q P* = (∀ *xys xy y* . (*xys*@[*xy*] ∈ *P* ∧ (∃ *q'* ∈ (*io-targets M xys q*) . [(*fst xy*,*y*)] ∈ *LS M q'*)) ⟶ *xys*@[(*fst xy*,*y*)] ∈ *P*)
⟨*proof*⟩

fun *output-complete-for-FSM-sequences-from-state'* :: ('*a*,'*b*,'*c*) *fsm* ⇒ '*a* ⇒ ('*b* × '*c*) *list set* ⇒ *bool* **where**
output-complete-for-FSM-sequences-from-state' M q P = (∀ *io*∈*P* . ∀ *t* ∈ *transitions M* . *io* = [] ∨ (*t-source t* ∈ *io-targets M* (*butlast io*) *q* ∧ *t-input t* = *fst* (*last io*) ⟶ (*butlast io*)@[(*fst* (*last io*), *t-output t*)] ∈ *P*))

lemma *output-complete-for-FSM-sequences-alt-def'*[*code*] : *output-complete-for-FSM-sequences-from-state*

M q P = output-complete-for-FSM-sequences-from-state′ M q P
⟨*proof*⟩

fun *deadlock-states-sequences* :: ('*a*,'*b*,'*c*) *fsm* ⇒ '*a set* ⇒ ('*b* × '*c*) *list set* ⇒ *bool*
where
deadlock-states-sequences M Q P = (∀ *xys* ∈ *P* .
((*io-targets M xys* (*initial M*) ⊆ *Q*
∧ ¬ (∃ *xys′* ∈ *P* . *length xys* < *length xys′* ∧ *take* (*length xys*) *xys′* = *xys*)))
∨ (¬ *io-targets M xys* (*initial M*) ∩ *Q* = {}
∧ (∃ *xys′* ∈ *P* . *length xys* < *length xys′* ∧ *take* (*length xys*) *xys′* = *xys*)))

fun *reachable-states-sequences* :: ('*a*,'*b*,'*c*) *fsm* ⇒ '*a set* ⇒ ('*b* × '*c*) *list set* ⇒ *bool*
where
reachable-states-sequences M Q P = (∀ *q* ∈ *Q* . ∃ *xys* ∈ *P* . *q* ∈ *io-targets M xys* (*initial M*))

fun *prefix-closed-sequences* :: ('*b* × '*c*) *list set* ⇒ *bool* **where**
prefix-closed-sequences P = (∀ *xys1 xys2* . *xys1*@*xys2* ∈ *P* ⟶ *xys1* ∈ *P*)

fun *prefix-closed-sequences′* :: ('*b* × '*c*) *list set* ⇒ *bool* **where**
prefix-closed-sequences′ P = (∀ *io* ∈ *P* . *io* = [] ∨ (*butlast io*) ∈ *P*)

lemma *prefix-closed-sequences-alt-def*[*code*] : *prefix-closed-sequences P* = *prefix-closed-sequences′ P*
⟨*proof*⟩

8.1 Completions

definition *prefix-completion* :: '*a list set* ⇒ '*a list set* **where**
prefix-completion P = {*xs* . ∃ *ys* . *xs*@*ys* ∈ *P*}

lemma *prefix-completion-closed* :
prefix-closed-sequences (*prefix-completion P*)
⟨*proof*⟩

lemma *prefix-completion-source-subset* :
P ⊆ *prefix-completion P*
⟨*proof*⟩

definition *output-completion-for-FSM* :: ('*a*,'*b*,'*c*) *fsm* ⇒ ('*b* × '*c*) *list set* ⇒ ('*b* × '*c*) *list set* **where**
output-completion-for-FSM M P = *P* ∪ { *io*@[(*x*,*y′*)] | *io x y′* . (*y′* ∈ (*outputs M*)) ∧ (∃ *y* . *io*@[(*x*,*y*)] ∈ *P*)}

lemma *output-completion-for-FSM-complete* :
shows *output-complete-sequences M* (*output-completion-for-FSM M P*)

⟨proof⟩

lemma *output-completion-for-FSM-length* :
assumes ∀ *io* ∈ *P* . *length io* ≤ *k*
shows ∀ *io* ∈ *output-completion-for-FSM M P*. *length io* ≤ *k*
⟨proof⟩

lemma *output-completion-for-FSM-code*[*code*] :
output-completion-for-FSM M P = *P* ∪ (⋃(*image* (λ(*y*,*io*) . *if length io* = *0 then* {} *else* {((*butlast io*)@[(*fst* (*last io*),*y*)])}) ((*outputs M*) × *P*)))
⟨proof⟩

end

9 Observability

This theory presents the classical algorithm for transforming FSMs into language-equivalent observable FSMs in analogy to the determinisation of nondeterministic finite automata.

theory *Observability*
imports *FSM*
begin

lemma *fPow-Pow* : *Pow* (*fset A*) = *fset* (*fset* |'| *fPow A*)
⟨proof⟩

lemma *fcard-fsubset*: ¬ *fcard* (*A* |−| (*B* |∪| *C*)) < *fcard* (*A* |−| *B*) ⟹ *C* |⊆| *A* ⟹ *C* |⊆| *B*
⟨proof⟩

lemma *make-observable-transitions-qtrans-helper*:
assumes *qtrans* = *ffUnion* (*fimage* (λ *q* . (*let qts* = *ffilter* (λ*t* . *t-source t* |∈| *q*) *A*;
ios = *fimage* (λ *t* . (*t-input t*, *t-output t*)) *qts*
in fimage (λ(*x*,*y*) . (*q*,*x*,*y*, *t-target* |'| ((*ffilter* (λ*t* . (*t-input t*, *t-output t*) = (*x*,*y*)) *qts*)))) *ios*)) *nexts*)
shows ⋀ *t* . *t* |∈| *qtrans* ⟷ *t-source t* |∈| *nexts* ∧ *t-target t* ≠ {||} ∧ *fset* (*t-target t*) = *t-target* ' {*t'* . *t'* |∈| *A* ∧ *t-source t'* |∈| *t-source t* ∧ *t-input t'* = *t-input t* ∧ *t-output t'* = *t-output t*}
⟨proof⟩

function *make-observable-transitions* :: ('*a*,'*b*,'*c*) *transition fset* ⇒ '*a fset fset* ⇒ '*a fset fset* ⇒ ('*a fset* × '*b* × '*c* × '*a fset*) *fset* ⇒ ('*a fset* × '*b* × '*c* × '*a fset*) *fset*

where
make-observable-transitions base-trans nexts dones ts = (*let*
qtrans = *ffUnion* (*fimage* (λ *q* . (*let qts* = *ffilter* (λ*t* . *t-source t* |∈| *q*) *base-trans*;
ios = *fimage* (λ *t* . (*t-input t*, *t-output t*)) *qts*
in fimage (λ(*x*,*y*) . (*q*,*x*,*y*, *t-target* |‘| (*ffilter* (λ*t* . (*t-input t*, *t-output t*) = (*x*,*y*)) *qts*))) *ios*)) *nexts*);
dones′ = *dones* |∪| *nexts*;
ts′ = *ts* |∪| *qtrans*;
nexts′ = (*fimage t-target qtrans*) |−| *dones*′
in if nexts′ = {||}
then ts′
else make-observable-transitions base-trans nexts′ *dones*′ *ts*′)
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *make-observable-transitions-mono*: *ts* |⊆| (*make-observable-transitions base-trans nexts dones ts*)
⟨*proof*⟩

inductive *pathlike* :: (′*state*, ′*input*, ′*output*) *transition fset* ⇒ ′*state* ⇒ (′*state*, ′*input*, ′*output*) *path* ⇒ *bool*
where
nil[*intro*!] : *pathlike ts q* [] |
cons[*intro*!] : *t* |∈| *ts* ⟹ *pathlike ts* (*t-target t*) *p* ⟹ *pathlike ts* (*t-source t*) (*t*#*p*)

inductive-cases *pathlike-nil-elim*[*elim*!]: *pathlike ts q* []
inductive-cases *pathlike-cons-elim*[*elim*!]: *pathlike ts q* (*t*#*p*)

lemma *make-observable-transitions-t-source* :
assumes ⋀ *t* . *t* |∈| *ts* ⟹ *t-source t* |∈| *dones* ∧ *t-target t* ≠ {||} ∧ *fset* (*t-target t*) = *t-target* ‘ {*t*′ . *t*′ |∈| *base-trans* ∧ *t-source t*′ |∈| *t-source t* ∧ *t-input t*′ = *t-input t* ∧ *t-output t*′ = *t-output t*}
and ⋀ *q t*′ . *q* |∈| *dones* ⟹ *t*′ |∈| *base-trans* ⟹ *t-source t*′ |∈| *q* ⟹ ∃ *t* . *t* |∈| *ts* ∧ *t-source t* = *q* ∧ *t-input t* = *t-input t*′ ∧ *t-output t* = *t-output t*′
and *t* |∈| *make-observable-transitions base-trans* ((*fimage t-target ts*) |−| *dones*) *dones ts*
and *t-source t* |∈| *dones*
shows *t* |∈| *ts*

⟨proof⟩

lemma *make-observable-transitions-path* :
assumes $\bigwedge$ *t* . *t* |∈| *ts* ⟹ *t-source t* |∈| *dones* ∧ *t-target t* ≠ {||} ∧ *fset* (*t-target t*) = *t-target* ‘ {*t'* ∈ *transitions M* . *t-source t'* |∈| *t-source t* ∧ *t-input t'* = *t-input t* ∧ *t-output t'* = *t-output t*}
and $\bigwedge$ *q t'* . *q* |∈| *dones* ⟹ *t'* ∈ *transitions M* ⟹ *t-source t'* |∈| *q* ⟹ ∃ *t* . *t* |∈| *ts* ∧ *t-source t* = *q* ∧ *t-input t* = *t-input t'* ∧ *t-output t* = *t-output t'*
and $\bigwedge$ *q* . *q* |∈| (*fimage t-target ts*) |−| *dones* ⟹ *q* |∈| *fPow* (*t-source* |‘| *ftransitions M* |∪| *t-target* |‘| *ftransitions M*)
and $\bigwedge$ *q* . *q* |∈| *dones* ⟹ *q* |∈| *fPow* (*t-source* |‘| *ftransitions M* |∪| *t-target* |‘| *ftransitions M* |∪| {|*initial M*|})
and {||} |∉| *dones*
and *q* |∈| *dones*
shows (∃ *q' p* . *q'* |∈| *q* ∧ *path M q' p* ∧ *p-io p* = *io*) ⟷ (∃ *p'*. *pathlike* (*make-observable-transitions* (*ftransitions M*) ((*fimage t-target ts*) |−| *dones*) *dones ts*) *q p'* ∧ *p-io p'* = *io*)

⟨proof⟩

fun *observable-fset* :: (*'a,'b,'c*) *transition fset* ⇒ *bool* **where**
observable-fset ts = (∀ *t1 t2* . *t1* |∈| *ts* ⟶ *t2* |∈| *ts* ⟶
t-source t1 = *t-source t2* ⟶ *t-input t1* = *t-input t2* ⟶ *t-output t1* = *t-output t2*
⟶ *t-target t1* = *t-target t2*)

lemma *make-observable-transitions-observable* :
assumes $\bigwedge$ *t* . *t* |∈| *ts* ⟹ *t-source t* |∈| *dones* ∧ *t-target t* ≠ {||} ∧ *fset* (*t-target t*) = *t-target* ‘ {*t'* . *t'* |∈| *base-trans* ∧ *t-source t'* |∈| *t-source t* ∧ *t-input t'* = *t-input t* ∧ *t-output t'* = *t-output t*}
and *observable-fset ts*
shows *observable-fset* (*make-observable-transitions base-trans* ((*fimage t-target ts*) |−| *dones*) *dones ts*)
⟨proof⟩

lemma *make-observable-transitions-transition-props* :
assumes $\bigwedge$ *t* . *t* |∈| *ts* ⟹ *t-source t* |∈| *dones* ∧ *t-target t* |∈| *dones* |∪| ((*fimage t-target ts*) |−| *dones*) ∧ *t-input t* |∈| *t-input* |'| *base-trans* ∧ *t-output t* |∈| *t-output* |'| *base-trans*
assumes *t* |∈| *make-observable-transitions base-trans* ((*fimage t-target ts*) |−| *dones*) *dones ts*
shows *t-source t* |∈| *dones* |∪| (*t-target* |'| (*make-observable-transitions base-trans* ((*fimage t-target ts*) |−| *dones*) *dones ts*))
and *t-target t* |∈| *dones* |∪| (*t-target* |'| (*make-observable-transitions base-trans* ((*fimage t-target ts*) |−| *dones*) *dones ts*))
and *t-input t* |∈| *t-input* |'| *base-trans*
and *t-output t* |∈| *t-output* |'| *base-trans*
⟨*proof*⟩

fun *make-observable* :: (*'a* :: *linorder*,*'b* :: *linorder*,*'c* :: *linorder*) *fsm* ⇒ (*'a fset*,*'b*,*'c*) *fsm* **where**
make-observable M = (*let*
initial-trans = (*let qts* = *ffilter* (λ*t* . *t-source t* = *initial M*) (*ftransitions M*);
ios = *fimage* (λ *t* . (*t-input t*, *t-output t*)) *qts*
in fimage (λ(*x*,*y*) . ({|*initial M*|},*x*,*y*, *t-target* |'| ((*ffilter* (λ*t* . (*t-input t*, *t-output t*) = (*x*,*y*)) *qts*)))) *ios*);
nexts = *fimage t-target initial-trans* |−| {|{|*initial M*|}|};
ptransitions = *make-observable-transitions* (*ftransitions M*) *nexts* {|{|*initial M*|}|} *initial-trans*;
pstates = *finsert* {|*initial M*|} (*t-target* |'| *ptransitions*);
M' = *create-unconnected-fsm-from-fsets* {|*initial M*|} *pstates* (*finputs M*) (*foutputs M*)
in add-transitions M' (*fset ptransitions*))

lemma *make-observable-language-observable* :
shows *L* (*make-observable M*) = *L M*
and *observable* (*make-observable M*)
and *initial* (*make-observable M*) = {|*initial M*|}
and *inputs* (*make-observable M*) = *inputs M*
and *outputs* (*make-observable M*) = *outputs M*
⟨*proof*⟩

end

10 Prefix Tree

This theory introduces a tree to efficiently store prefix-complete sets of lists. Several functions to lookup or merge subtrees are provided.

theory *Prefix-Tree*
imports *Util HOL−Library.Mapping HOL−Library.List-Lexorder*
begin

datatype *'a prefix-tree* = *PT 'a ⇀ 'a prefix-tree*

definition *empty* :: *'a prefix-tree* **where**
empty = *PT Map.empty*

fun *isin* :: *'a prefix-tree* ⇒ *'a list* ⇒ *bool* **where**
isin t [] = *True* |
isin (*PT m*) (*x* # *xs*) = (*case m x of None* ⇒ *False* | *Some t* ⇒ *isin t xs*)

lemma *isin-prefix* :
assumes *isin t* (*xs*@*xs'*)
shows *isin t xs*
⟨*proof*⟩

fun *set* :: *'a prefix-tree* ⇒ *'a list set* **where**
set t = {*xs* . *isin t xs*}

lemma *set-empty* : *set empty* = ({[]} :: *'a list set*)
⟨*proof*⟩

lemma *set-Nil* : [] ∈ *set t*
⟨*proof*⟩

fun *insert* :: *'a prefix-tree* ⇒ *'a list* ⇒ *'a prefix-tree* **where**
insert t [] = *t* |
insert (*PT m*) (*x*#*xs*) = *PT* (*m*(*x* ↦ *insert* (*case m x of None* ⇒ *empty* | *Some t'* ⇒ *t'*) *xs*))

lemma *insert-isin-prefix* : *isin* (*insert t* (*xs*@*xs'*)) *xs*
⟨*proof*⟩

lemma *insert-isin-other* :
assumes *isin t xs*
shows *isin* (*insert t xs'*) *xs*
⟨*proof*⟩

lemma *insert-isin-rev* :
assumes *isin* (*insert t xs'*) *xs*
shows *isin t xs* ∨ (∃ *xs''* . *xs'* = *xs*@*xs''*)
⟨*proof*⟩

lemma *insert-set* : *set* (*insert t xs*) = *set t* ∪ {*xs'* . ∃ *xs''* . *xs* = *xs'*@*xs''*}
⟨*proof*⟩

lemma *insert-isin* : *xs* ∈ *set* (*insert t xs*)
⟨*proof*⟩

lemma *set-prefix* :
assumes *xs*@*ys* ∈ *set T*
shows *xs* ∈ *set T*
⟨*proof*⟩

fun *after* :: *'a prefix-tree* ⇒ *'a list* ⇒ *'a prefix-tree* **where**
after t [] = *t* |
after (*PT m*) (*x* # *xs*) = (*case m x of None* ⇒ *empty* | *Some t* ⇒ *after t xs*)

lemma *after-set* : *set* (*after t xs*) = *Set.insert* [] {*xs'* . *xs*@*xs'* ∈ *set t*}
(**is** *?A t xs* = *?B t xs*)
⟨*proof*⟩

lemma *after-set-Cons* :
assumes $\gamma \in$ *set* (*after T* α)
and $\gamma \neq []$
shows $\alpha \in$ *set T*
⟨*proof*⟩

function (*domintros*) *combine* :: *'a prefix-tree* ⇒ *'a prefix-tree* ⇒ *'a prefix-tree*
where
combine (*PT m1*) (*PT m2*) = (*PT* (λ *x* . *case m1 x of*
None ⇒ *m2 x* |
Some t1 ⇒ (*case m2 x of*
None ⇒ *Some t1* |
Some t2 ⇒ *Some* (*combine t1 t2*))))
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *combine-alt-def* :
combine (*PT m1*) (*PT m2*) = *PT* (λ*x* . *combine-options combine* (*m1 x*) (*m2 x*))

⟨*proof*⟩

lemma *combine-set* :
set (*combine t1 t2*) = *set t1* ∪ *set t2*
⟨*proof*⟩

fun *combine-after* :: *'a prefix-tree* ⇒ *'a list* ⇒ *'a prefix-tree* ⇒ *'a prefix-tree* **where**
combine-after t1 [] *t2* = *combine t1 t2* |
combine-after (*PT m*) (*x#xs*) *t2* = *PT* (*m*(*x* ↦ *combine-after* (*case m x of None* ⇒ *empty* | *Some t'* ⇒ *t'*) *xs t2*))

lemma *combine-after-set* : *set* (*combine-after t1 xs t2*) = *set t1* ∪ {*xs'* . ∃ *xs''* . *xs* = *xs'*@*xs''*} ∪ {*xs*@*xs'* | *xs'* . *xs'* ∈ *set t2*}
⟨*proof*⟩

fun *from-list* :: *'a list list* ⇒ *'a prefix-tree* **where**
from-list xs = *foldr* (λ *x t* . *insert t x*) *xs empty*

lemma *from-list-set* : *set* (*from-list xs*) = *Set.insert* [] {*xs''* . ∃ *xs' xs'''* . *xs'* ∈ *list.set xs* ∧ *xs'* = *xs''*@*xs'''*}
⟨*proof*⟩

lemma *from-list-subset* : *list.set xs* ⊆ *set* (*from-list xs*)
⟨*proof*⟩

lemma *from-list-set-elem* :
assumes *x* ∈ *list.set xs*
shows *x* ∈ *set* (*from-list xs*)
⟨*proof*⟩

function (*domintros*) *finite-tree* :: *'a prefix-tree* ⇒ *bool* **where**
finite-tree (*PT m*) = (*finite* (*dom m*) ∧ (∀ *t* ∈ *ran m* . *finite-tree t*))
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *combine-after-after-subset* :
set T2 ⊆ *set* (*after* (*combine-after T1 xs T2*) *xs*)
⟨*proof*⟩

lemma *subset-after-subset* :
set T2 ⊆ *set T1* ⟹ *set* (*after T2 xs*) ⊆ *set* (*after T1 xs*)
⟨*proof*⟩

lemma *set-alt-def* :
set (*PT m*) = *Set.insert* [] ($\bigcup$ *x* ∈ *dom m* . (*Cons x*) ‘ (*set* (*the* (*m x*))))
(**is** *?A m* = *?B m*)
⟨*proof*⟩

lemma *finite-tree-iff* :
finite-tree t = *finite* (*set t*)
(**is** *?P1* = *?P2*)
⟨*proof*⟩

lemma *empty-finite-tree* :
finite-tree empty
⟨*proof*⟩

lemma *insert-finite-tree* :
assumes *finite-tree t*
shows *finite-tree* (*insert t xs*)
⟨*proof*⟩

lemma *from-list-finite-tree* :
finite-tree (*from-list xs*)
⟨*proof*⟩

lemma *combine-after-finite-tree* :
assumes *finite-tree t1*
and *finite-tree t2*
shows *finite-tree* (*combine-after t1 α t2*)
⟨*proof*⟩

lemma *combine-finite-tree* :
assumes *finite-tree t1*
and *finite-tree t2*
shows *finite-tree* (*combine t1 t2*)
⟨*proof*⟩

function (*domintros*) *sorted-list-of-maximal-sequences-in-tree* :: (*'a* :: *linorder*) *prefix-tree* ⇒ *'a list list* **where**
sorted-list-of-maximal-sequences-in-tree (*PT m*) =
(*if dom m* = {}
then [[]]
else concat (*map* (*λk* . *map* ((#) *k*) (*sorted-list-of-maximal-sequences-in-tree* (*the* (*m k*)))) (*sorted-list-of-set* (*dom m*))))
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *sorted-list-of-maximal-sequences-in-tree-Nil* :
assumes $[] \in$ *list.set* (*sorted-list-of-maximal-sequences-in-tree t*)
shows $t = empty$
⟨*proof*⟩

lemma *sorted-list-of-maximal-sequences-in-tree-set* :
assumes *finite-tree t*
shows *list.set* (*sorted-list-of-maximal-sequences-in-tree t*) $= \{y.\ y \in set\ t \land \neg(\exists y'.\ y' \neq [] \land y@y' \in set\ t)\}$
(**is** *?S1* = *?S2*)
⟨*proof*⟩

lemma *sorted-list-of-maximal-sequences-in-tree-ob* :
assumes *finite-tree T*
and $xs \in set\ T$
obtains xs' **where** $xs@xs' \in$ *list.set* (*sorted-list-of-maximal-sequences-in-tree T*)
⟨*proof*⟩

function (*domintros*) *sorted-list-of-sequences-in-tree* :: (*'a* :: *linorder*) *prefix-tree* ⇒ *'a list list* **where**
sorted-list-of-sequences-in-tree (*PT m*) =
(*if dom m* = {}
then [[]]
else [] # *concat* (*map* (λ*k* . *map* ((#) *k*) (*sorted-list-of-sequences-in-tree* (*the* (*m k*)))) (*sorted-list-of-set* (*dom m*))))
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *sorted-list-of-sequences-in-tree-set* :
assumes *finite-tree t*
shows *list.set* (*sorted-list-of-sequences-in-tree t*) = *set t*
(**is** *?S1* = *?S2*)
⟨*proof*⟩

fun *difference-list* :: (*'a*::*linorder*) *prefix-tree* ⇒ *'a prefix-tree* ⇒ *'a list list* **where**
difference-list t1 t2 = *filter* (λ *xs* . ¬ *isin t2 xs*) (*sorted-list-of-sequences-in-tree t1*)

lemma *difference-list-set* :
assumes *finite-tree t1*
shows *List.set* (*difference-list t1 t2*) = (*set t1* − *set t2*)

⟨*proof*⟩

fun *is-leaf* :: *'a prefix-tree* ⇒ *bool* **where**
is-leaf t = (*t* = *empty*)

fun *is-maximal-in* :: *'a prefix-tree* ⇒ *'a list* ⇒ *bool* **where**
is-maximal-in T α = (*isin T α* ∧ *is-leaf* (*after T α*))

function (*domintros*) *height* :: *'a prefix-tree* ⇒ *nat* **where**
height (*PT m*) = (*if* (*is-leaf* (*PT m*)) *then 0 else 1* + *Max* (*height* ‘ *ran m*))
⟨*proof*⟩
termination
⟨*proof*⟩

function (*domintros*) *height-over* :: *'a list* ⇒ *'a prefix-tree* ⇒ *nat* **where**
height-over xs (*PT m*) = *1* + *foldr* (*λ x maxH* . *case m x of Some t'* ⇒ *max* (*height-over xs t'*) *maxH* | *None* ⇒ *maxH*) *xs 0*
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *height-over-empty* :
height-over xs empty = *1*
⟨*proof*⟩

lemma *height-over-subtree-less* :
assumes *m x* = *Some t'*
and *x* ∈ *list.set xs*
shows *height-over xs t'* < *height-over xs* (*PT m*)
⟨*proof*⟩

fun *maximum-prefix* :: *'a prefix-tree* ⇒ *'a list* ⇒ *'a list* **where**
maximum-prefix t [] = [] |
maximum-prefix (*PT m*) (*x* # *xs*) = (*case m x of None* ⇒ [] | *Some t* ⇒ *x* # *maximum-prefix t xs*)

lemma *maximum-prefix-isin* :
isin t (*maximum-prefix t xs*)
⟨*proof*⟩

lemma *maximum-prefix-maximal* :
maximum-prefix t xs = *xs*
∨ (∃ *x' xs'* . *xs* = (*maximum-prefix t xs*)@[*x'*]@*xs'* ∧ ¬ *isin t* ((*maximum-prefix t xs*)@[*x'*]))
⟨*proof*⟩

fun *maximum-fst-prefixes* :: ('a×'b) *prefix-tree* ⇒ 'a *list* ⇒ 'b *list* ⇒ ('a×'b) *list list* **where**
maximum-fst-prefixes t [] *ys* = (*if is-leaf t then* [[]] *else* []) |
maximum-fst-prefixes (*PT m*) (*x* # *xs*) *ys* = (*if is-leaf* (*PT m*) *then* [[]] *else concat* (*map* (λ *y* . *map* ((#) (*x*,*y*)) (*maximum-fst-prefixes* (*the* (*m* (*x*,*y*))) *xs ys*)) (*filter* (λ *y* . (*m* (*x*,*y*) ≠ *None*)) *ys*)))

lemma *maximum-fst-prefixes-set* :
list.set (*maximum-fst-prefixes t xs ys*) ⊆ *set t*
⟨*proof*⟩

lemma *maximum-fst-prefixes-are-prefixes* :
assumes *xys* ∈ *list.set* (*maximum-fst-prefixes t xs ys*)
shows *map fst xys* = *take* (*length xys*) *xs*
⟨*proof*⟩

lemma *finite-tree-set-eq* :
assumes *set t1* = *set t2*
and *finite-tree t1*
shows *t1* = *t2*
⟨*proof*⟩

fun *after-fst* :: ('a × 'b) *prefix-tree* ⇒ 'a *list* ⇒ 'b *list* ⇒ ('a × 'b) *prefix-tree* **where**
after-fst t [] *ys* = *t* |
after-fst (*PT m*) (*x* # *xs*) *ys* = *foldr* (λ *y t* . *case m* (*x*,*y*) *of None* ⇒ *t* | *Some t'* ⇒ *combine t* (*after-fst t' xs ys*)) *ys empty*

10.1 Alternative characterization for code generation

In order to generate code for the prefix trees, we represent the map inside each prefix tree by Mapping.

definition *MPT* :: ('a,'a *prefix-tree*) *mapping* ⇒ 'a *prefix-tree* **where**
MPT m = *PT* (*Mapping.lookup m*)

code-datatype *MPT*

lemma *equals-MPT*[*code*]: *equal-class.equal* (*MPT m1*) (*MPT m2*) = (*m1* = *m2*)

⟨*proof*⟩

lemma *empty-MPT*[*code*] :
  empty = *MPT Mapping.empty*
  ⟨*proof*⟩

lemma *insert-MPT*[*code*] :
  insert (*MPT m*) *xs* = (*case xs of*
    [] ⇒ (*MPT m*) |
    (*x#xs*) ⇒ *MPT* (*Mapping.update x* (*insert* (*case Mapping.lookup m x of None* ⇒ *empty* | *Some t′* ⇒ *t′*) *xs*) *m*))
  ⟨*proof*⟩

lemma *isin-MPT*[*code*] :
  isin (*MPT m*) *xs* = (*case xs of*
    [] ⇒ *True* |
    (*x#xs*) ⇒ (*case Mapping.lookup m x of None* ⇒ *False* | *Some t* ⇒ *isin t xs*))
  ⟨*proof*⟩

lemma *after-MPT*[*code*] :
  after (*MPT m*) *xs* = (*case xs of*
    [] ⇒ *MPT m* |
    (*x#xs*) ⇒ (*case Mapping.lookup m x of None* ⇒ *empty* | *Some t* ⇒ *after t xs*))
  ⟨*proof*⟩

lemma *PT-Mapping-ob* :
  fixes *t* :: *′a prefix-tree*
  obtains *m* **where** *t* = *MPT m*
⟨*proof*⟩

lemma *set-MPT*[*code*] :
  set (*MPT m*) = *Set.insert* [] (⋃ *x* ∈ *Mapping.keys m* . (*Cons x*) ‘ (*set* (*the* (*Mapping.lookup m x*))))
  ⟨*proof*⟩

lemma *combine-MPT*[*code*] :
  combine (*MPT m1*) (*MPT m2*) = *MPT* (*Mapping.combine combine m1 m2*)
⟨*proof*⟩

lemma *combine-after-MPT*[*code*] :
  combine-after (*MPT m*) *xs t* = (*case xs of*
    [] ⇒ *combine* (*MPT m*) *t* |
    (*x#xs*) ⇒ *MPT* (*Mapping.update x* (*combine-after* (*case Mapping.lookup m x of None* ⇒ *empty* | *Some t′* ⇒ *t′*) *xs t*) *m*))
  ⟨*proof*⟩

lemma *finite-tree-MPT*[*code*] :
finite-tree (*MPT m*) = (*finite* (*Mapping.keys m*) ∧ (∀ *x* ∈ *Mapping.keys m* . *finite-tree* (*the* (*Mapping.lookup m x*))))
⟨*proof*⟩

lemma *sorted-list-of-maximal-sequences-in-tree-MPT*[*code*] :
sorted-list-of-maximal-sequences-in-tree (*MPT m*) =
(*if Mapping.keys m* = {}
then [[]]
else concat (*map* (λ*k* . *map* ((#) *k*) (*sorted-list-of-maximal-sequences-in-tree* (*the* (*Mapping.lookup m k*)))) (*sorted-list-of-set* (*Mapping.keys m*))))
⟨*proof*⟩

lemma *is-leaf-MPT*[*code*]:
is-leaf (*MPT m*) = (*Mapping.is-empty m*)
⟨*proof*⟩

lemma *height-MPT*[*code*] :
height (*MPT m*) = (*if* (*is-leaf* (*MPT m*)) *then 0 else 1* + *Max* ((*height* ∘ *the* ∘ *Mapping.lookup m*) ' *Mapping.keys m*))
⟨*proof*⟩

lemma *maximum-prefix-MPT*[*code*]:
maximum-prefix (*MPT m*) *xs* = (*case xs of*
[] ⇒ [] |
(*x*#*xs*) ⇒ (*case Mapping.lookup m x of None* ⇒ [] | *Some t* ⇒ *x* # *maximum-prefix t xs*))
⟨*proof*⟩

lemma *sorted-list-of-in-tree-MPT*[*code*] :
sorted-list-of-sequences-in-tree (*MPT m*) =
(*if Mapping.keys m* = {}
then [[]]
else [] # *concat* (*map* (λ*k* . *map* ((#) *k*) (*sorted-list-of-sequences-in-tree* (*the* (*Mapping.lookup m k*)))) (*sorted-list-of-set* (*Mapping.keys m*))))
⟨*proof*⟩

lemma *maximum-fst-prefixes-leaf*:
fixes *xs* :: ′*a list* **and** *ys* :: ′*b list*
shows *maximum-fst-prefixes empty xs ys* = [[]]
⟨*proof*⟩

lemma *maximum-fst-prefixes-MPT*[*code*]:
maximum-fst-prefixes (*MPT m*) *xs ys* = (*case xs of*
[] ⇒ (*if is-leaf* (*MPT m*) *then* [[]] *else* []) |
(*x* # *xs*) ⇒ (*if is-leaf* (*MPT m*) *then* [[]] *else concat* (*map* (λ *y* . *map* ((#)

(*x*,*y*)) (*maximum-fst-prefixes* (*the* (*Mapping.lookup m* (*x*,*y*))) *xs ys*)) (*filter* (λ *y* . (*Mapping.lookup m* (*x*,*y*) ≠ *None*)) *ys*))))
⟨*proof*⟩

end

11 Refined Code Generation for Prefix Trees

This theory provides alternative code equations for selected functions on prefix trees. Currently only Mapping via RBT is supported.

theory *Prefix-Tree-Refined*
imports *Prefix-Tree Containers.Containers*
begin

declare [[*code drop*: *Prefix-Tree.combine*]]

lemma *combine-refined*[*code*] :
fixes *m1* :: (′*a* :: *ccompare*, ′*a prefix-tree*) *mapping-rbt*
shows *Prefix-Tree.combine* (*MPT* (*RBT-Mapping m1*)) (*MPT* (*RBT-Mapping m2*))
= (*case ID CCOMPARE*(′*a*) *of*
None ⇒ *Code.abort* (*STR* ″*combine-MPT-RBT-Mapping*: *ccompare* = *None*″) (λ-. *Prefix-Tree.combine* (*MPT* (*RBT-Mapping m1*)) (*MPT* (*RBT-Mapping m2*)))
| *Some* - ⇒ *MPT* (*RBT-Mapping* (*RBT-Mapping2.join* (λ *a t1 t2* . *Prefix-Tree.combine t1 t2*) *m1 m2*)))
(**is** *?PT1* = *?PT2*)
⟨*proof*⟩

declare [[*code drop*: *Prefix-Tree.is-leaf*]]

lemma *is-leaf-refined*[*code*] :
fixes *m* :: (′*a* :: *ccompare*, ′*a prefix-tree*) *mapping-rbt*
shows *Prefix-Tree.is-leaf* (*MPT* (*RBT-Mapping m*))
= (*case ID CCOMPARE*(′*a*) *of*
None ⇒ *Code.abort* (*STR* ″*is-leaf-MPT-RBT-Mapping*: *ccompare* =

None'') (λ-. *Prefix-Tree.is-leaf* (*MPT* (*RBT-Mapping m*)))
| *Some* - ⇒ *RBT-Mapping2.is-empty m*)
(**is** *?PT1* = *?PT2*)
⟨*proof*⟩

end

12 State Cover

This theory introduces a simple depth-first strategy for computing state covers.

theory *State-Cover*
imports *FSM*
begin

12.1 Basic Definitions

type-synonym (*'a,'b*) *state-cover* = (*'a* × *'b*) *list set*
type-synonym (*'a,'b,'c*) *state-cover-assignment* = *'a* ⇒ (*'b* × *'c*) *list*

fun *is-state-cover* :: (*'a,'b,'c*) *fsm* ⇒ (*'b,'c*) *state-cover* ⇒ *bool* **where**
is-state-cover M SC = ([] ∈ *SC* ∧ (∀ *q* ∈ *reachable-states M* . ∃ *io* ∈ *SC* . *q* ∈ *io-targets M io* (*initial M*)))

fun *is-state-cover-assignment* :: (*'a,'b,'c*) *fsm* ⇒ (*'a,'b,'c*) *state-cover-assignment* ⇒ *bool* **where**
is-state-cover-assignment M f = (*f* (*initial M*) = [] ∧ (∀ *q* ∈ *reachable-states M* . *q* ∈ *io-targets M* (*f q*) (*initial M*)))

lemma *state-cover-assignment-from-state-cover* :
assumes *is-state-cover M SC*
obtains *f* **where** *is-state-cover-assignment M f*
and ⋀ *q* . *q* ∈ *reachable-states M* ⟹ *f q* ∈ *SC*
⟨*proof*⟩

lemma *is-state-cover-assignment-language* :
assumes *is-state-cover-assignment M V*
and *q* ∈ *reachable-states M*
shows *V q* ∈ *L M*
⟨*proof*⟩

lemma *is-state-cover-assignment-observable-after* :
assumes *observable M*
and *is-state-cover-assignment M V*
and *q* ∈ *reachable-states M*
shows *after-initial M* (*V q*) = *q*
⟨*proof*⟩

lemma *non-initialized-state-cover-assignment-from-non-initialized-state-cover* :
assumes $\bigwedge$ *q* . *q* ∈ *reachable-states M* ⟹ ∃ *io* ∈ *L M* ∩ *SC* . *q* ∈ *io-targets M io* (*initial M*)
obtains *f* **where** $\bigwedge$ *q* . *q* ∈ *reachable-states M* ⟹ *q* ∈ *io-targets M* (*f q*) (*initial M*)
and $\bigwedge$ *q* . *q* ∈ *reachable-states M* ⟹ *f q* ∈ *L M* ∩ *SC*
⟨*proof*⟩

lemma *state-cover-assignment-inj* :
assumes *is-state-cover-assignment M V*
and *observable M*
and *q1* ∈ *reachable-states M*
and *q2* ∈ *reachable-states M*
and *q1* ≠ *q2*
shows *V q1* ≠ *V q2*
⟨*proof*⟩

lemma *state-cover-assignment-card* :
assumes *is-state-cover-assignment M V*
and *observable M*
shows *card* (*V* ‘ *reachable-states M*) = *card* (*reachable-states M*)
⟨*proof*⟩

lemma *state-cover-assignment-language* :
assumes *is-state-cover-assignment M V*
shows *V* ‘ *reachable-states M* ⊆ *L M*
⟨*proof*⟩

fun *is-minimal-state-cover* :: (′*a*,′*b*,′*c*) *fsm* ⇒ (′*b*,′*c*) *state-cover* ⇒ *bool* **where**
is-minimal-state-cover M SC = (∃ *f* . (*SC* = *f* ‘ *reachable-states M*) ∧ (*is-state-cover-assignment M f*))

lemma *minimal-state-cover-is-state-cover* :
assumes *is-minimal-state-cover M SC*
shows *is-state-cover M SC*
⟨*proof*⟩

lemma *state-cover-assignment-after* :
assumes *observable M*
and *is-state-cover-assignment M V*
and *q* ∈ *reachable-states M*
shows *V q* ∈ *L M* **and** *after-initial M* (*V q*) = *q*
⟨*proof*⟩

definition *covered-transitions* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $('a,'b,'c)$ *state-cover-assignment* $\Rightarrow$ $('b \times 'c)$ *list* $\Rightarrow$ $('a,'b,'c)$ *transition set* **where**
covered-transitions M V α = (*let*
ts = *the-elem* (*paths-for-io M* (*initial M*) α)
in
List.set (*filter* (λt . ((*V* (*t-source t*)) @ [(*t-input t*, *t-output t*)]) = (*V* (*t-target t*))) *ts*))

12.2 State Cover Computation

fun *reaching-paths-up-to-depth* :: $('a::linorder,'b::linorder,'c::linorder)$ *fsm* $\Rightarrow$ *'a set* $\Rightarrow$ *'a set* $\Rightarrow$ $('a \Rightarrow ('a,'b,'c)$ *path option*$)$ $\Rightarrow$ *nat* $\Rightarrow$ $('a \Rightarrow ('a,'b,'c)$ *path option*$)$
where
reaching-paths-up-to-depth M nexts dones assignment 0 = *assignment* |
reaching-paths-up-to-depth M nexts dones assignment (*Suc k*) = (*let*
usable-transitions = *filter* (λ *t* . *t-source t* $\in$ *nexts* $\wedge$ *t-target t* $\notin$ *dones* $\wedge$ *t-target t* $\notin$ *nexts*) (*transitions-as-list M*);
targets = *map t-target usable-transitions*;
transition-choice = *Map.empty*(*targets* [$\mapsto$] *usable-transitions*);
assignment' = *assignment*(*targets* [$\mapsto$] (*map* ($\lambda q'$. *case transition-choice q' of Some t* $\Rightarrow$ (*case assignment* (*t-source t*) *of Some p* $\Rightarrow$ *p*@[*t*])) *targets*));
nexts' = *set targets*;
dones' = *nexts* $\cup$ *dones*
in reaching-paths-up-to-depth M nexts' dones' assignment' k)

lemma *reaching-paths-up-to-depth-set* :
assumes *nexts* = {*q* . ($\exists$ *p* . *path M* (*initial M*) *p* $\wedge$ *target* (*initial M*) *p* = *q* $\wedge$ *length p* = *n*) $\wedge$ ($\nexists$ *p* . *path M* (*initial M*) *p* $\wedge$ *target* (*initial M*) *p* = *q* $\wedge$ *length p* < *n*)}
and *dones* = {*q* . $\exists$ *p* . *path M* (*initial M*) *p* $\wedge$ *target* (*initial M*) *p* = *q* $\wedge$ *length p* < *n*}
and $\bigwedge$ *q* . *assignment q* = *None* = ($\nexists$ *p* . *path M* (*initial M*) *p* $\wedge$ *target* (*initial M*) *p* = *q* $\wedge$ *length p* $\leq$ *n*)
and $\bigwedge$ *q p* . *assignment q* = *Some p* $\Longrightarrow$ *path M* (*initial M*) *p* $\wedge$ *target* (*initial M*) *p* = *q* $\wedge$ *length p* $\leq$ *n*
and *dom assignment* = *nexts* $\cup$ *dones*
shows ((*reaching-paths-up-to-depth M nexts dones assignment k*) *q* = *None*) = ($\nexists$ *p* . *path M* (*initial M*) *p* $\wedge$ *target* (*initial M*) *p* = *q* $\wedge$ *length p* $\leq$ *n*+*k*)
and ((*reaching-paths-up-to-depth M nexts dones assignment k*) *q* = *Some p*) $\Longrightarrow$ *path M* (*initial M*) *p* $\wedge$ *target* (*initial M*) *p* = *q* $\wedge$ *length p* $\leq$ *n*+*k*
and *q* $\in$ *nexts* $\cup$ *dones* $\Longrightarrow$ (*reaching-paths-up-to-depth M nexts dones assignment k*) *q* = *assignment q*
⟨*proof*⟩

fun *get-state-cover-assignment* :: $('a::linorder,'b::linorder,'c::linorder)$ *fsm* $\Rightarrow$ $('a,'b,'c)$

state-cover-assignment **where**
get-state-cover-assignment M = (*let*
path-assignments = *reaching-paths-up-to-depth M* {*initial M*} {} [*initial M* ↦ []] (*size M* −*1*)
in (λ *q* . *case path-assignments q of Some p* ⇒ *p-io p* | *None* ⇒ []))

lemma *get-state-cover-assignment-is-state-cover-assignment* :
is-state-cover-assignment M (*get-state-cover-assignment M*)
⟨*proof*⟩

12.3 Computing Reachable States via State Cover Computation

lemma *restrict-to-reachable-states*[*code*]:
restrict-to-reachable-states M = (*let*
path-assignments = *reaching-paths-up-to-depth M* {*initial M*} {} [*initial M* ↦ []] (*size M* −*1*)
in filter-states M (λ *q* . *path-assignments q* ≠ *None*))
⟨*proof*⟩

declare [[*code drop*: *reachable-states*]]
lemma *reachable-states-refined*[*code*] :
reachable-states M = (*let*
path-assignments = *reaching-paths-up-to-depth M* {*initial M*} {} [*initial M* ↦ []] (*size M* −*1*)
in Set.filter (λ *q* . *path-assignments q* ≠ *None*) (*states M*))
⟨*proof*⟩

lemma *minimal-sequence-to-failure-from-state-cover-assignment-ob* :
assumes *L M* ≠ *L I*
and *is-state-cover-assignment M V*
and (*L M* ∩ (*V* ‘ *reachable-states M*)) = (*L I* ∩ (*V* ‘ *reachable-states M*))
obtains *ioT ioX* **where** *ioT* ∈ (*V* ‘ *reachable-states M*)
and *ioT* @ *ioX* ∈ (*L M* − *L I*) ∪ (*L I* − *L M*)
and ⋀ *io q* . *q* ∈ *reachable-states M* ⟹ (*V q*)@*io* ∈ (*L M* − *L I*) ∪ (*L I* − *L M*) ⟹ *length ioX* ≤ *length io*
⟨*proof*⟩

end

13 Alternative OFSM Table Computation

The approach to computing OFSM tables presented in the imported theories is easy to use in proofs but inefficient in practice due to repeated recomputation of the same tables. Thus, in the following we present a more efficient method for computing and storing tables.

theory *OFSM-Tables-Refined*
imports *Minimisation Distinguishability*
begin

13.1 Computing a List of all OFSM Tables

type-synonym ('a,'b,'c) *ofsm-table* = ('a, 'a *set*) *mapping*

fun *initial-ofsm-table* :: ('a::*linorder*,'b,'c) *fsm* ⇒ ('a,'b,'c) *ofsm-table* **where**
initial-ofsm-table M = *Mapping.tabulate* (*states-as-list M*) (λ*q* . *states M*)

abbreviation *ofsm-lookup* ≡ *Mapping.lookup-default* {}

lemma *initial-ofsm-table-lookup-invar*: *ofsm-lookup* (*initial-ofsm-table M*) *q* = *ofsm-table M* (λ*q* . *states M*) *0 q*
⟨*proof*⟩

lemma *initial-ofsm-table-keys-invar*: *Mapping.keys* (*initial-ofsm-table M*) = *states M*
⟨*proof*⟩

fun *next-ofsm-table* :: ('a::*linorder*,'b,'c) *fsm* ⇒ ('a,'b,'c) *ofsm-table* ⇒ ('a,'b,'c) *ofsm-table* **where**
next-ofsm-table M prev-table = *Mapping.tabulate* (*states-as-list M*) (λ *q* . {*q'* ∈ *ofsm-lookup prev-table q* . ∀ *x* ∈ *inputs M* . ∀ *y* ∈ *outputs M* . (*case h-obs M q x y of Some qT* ⇒ (*case h-obs M q' x y of Some qT'* ⇒ *ofsm-lookup prev-table qT* = *ofsm-lookup prev-table qT'* | *None* ⇒ *False*) | *None* ⇒ *h-obs M q' x y* = *None*) })

lemma *h-obs-non-state* :
assumes *q* ∉ *states M*
shows *h-obs M q x y* = *None*
⟨*proof*⟩

lemma *next-ofsm-table-lookup-invar*:
assumes ⋀ *q* . *ofsm-lookup prev-table q* = *ofsm-table M* (λ*q* . *states M*) *k q*
shows *ofsm-lookup* (*next-ofsm-table M prev-table*) *q* = *ofsm-table M* (λ*q* . *states M*) (*Suc k*) *q*
⟨*proof*⟩

lemma *next-ofsm-table-keys-invar*: *Mapping.keys* (*next-ofsm-table M prev-table*) = *states M*
⟨*proof*⟩

fun *compute-ofsm-table-list* :: ('*a*::*linorder*,'*b*,'*c*) *fsm* ⇒ *nat* ⇒ ('*a*,'*b*,'*c*) *ofsm-table list* **where**
compute-ofsm-table-list M k = *rev* (*foldr* (λ - *prev* . (*next-ofsm-table M* (*hd prev*)) # *prev*) [*0*..<*k*] [*initial-ofsm-table M*])

lemma *compute-ofsm-table-list-props*:
length (*compute-ofsm-table-list M k*) = *Suc k*
⋀ *i q* . *i* < *Suc k* ⟹ *ofsm-lookup* ((*compute-ofsm-table-list M k*) ! *i*) *q* = *ofsm-table M* (λ*q* . *states M*) *i q*
⋀ *i* . *i* < *Suc k* ⟹ *Mapping.keys* ((*compute-ofsm-table-list M k*) ! *i*) = *states M*
⟨*proof*⟩

fun *compute-ofsm-tables* :: ('*a*::*linorder*,'*b*,'*c*) *fsm* ⇒ *nat* ⇒ (*nat*, ('*a*,'*b*,'*c*) *ofsm-table*) *mapping* **where**
compute-ofsm-tables M k = *Mapping.bulkload* (*compute-ofsm-table-list M k*)

lemma *compute-ofsm-tables-entries* :
assumes *i* < *Suc k*
shows (*the* (*Mapping.lookup* (*compute-ofsm-tables M k*) *i*)) = ((*compute-ofsm-table-list M k*) ! *i*)
⟨*proof*⟩

lemma *compute-ofsm-tables-lookup-invar* :
assumes *i* < *Suc k*
shows *ofsm-lookup* (*the* (*Mapping.lookup* (*compute-ofsm-tables M k*) *i*)) *q* = *ofsm-table M* (λ*q* . *states M*) *i q*
⟨*proof*⟩

lemma *compute-ofsm-tables-keys-invar* :
assumes *i* < *Suc k*
shows *Mapping.keys* (*the* (*Mapping.lookup* (*compute-ofsm-tables M k*) *i*)) = *states M*
⟨*proof*⟩

13.2 Finding Diverging Tables

lemma *ofsm-table-fix-from-compute-ofsm-tables* :
assumes *q* ∈ *states M*
shows *ofsm-lookup* (*the* (*Mapping.lookup* (*compute-ofsm-tables M* (*size M* − *1*)) (*size M* − *1*))) *q* = *ofsm-table-fix M* (λ*q*. *FSM.states M*) *0 q*

⟨*proof*⟩

fun *find-first-distinct-ofsm-table′* :: (′*a*::*linorder*,′*b*,′*c*) *fsm* ⇒ ′*a* ⇒ ′*a* ⇒ *nat* **where**
find-first-distinct-ofsm-table′ M q1 q2 = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*))
in if (*q1* ∈ *states M*
∧ *q2* ∈ *states M*
∧ (*ofsm-lookup* (*the* (*Mapping.lookup tables* (*size M* − *1*))) *q1*
≠ *ofsm-lookup* (*the* (*Mapping.lookup tables* (*size M* − *1*))) *q2*))
then the (*find-index* (λ *i* . *ofsm-lookup* (*the* (*Mapping.lookup tables i*)) *q1* ≠ *ofsm-lookup* (*the* (*Mapping.lookup tables i*)) *q2*) [*0*..<*size M*])
else 0)

lemma *find-first-distinct-ofsm-table-is-first′* :
assumes *q1* ∈ *FSM.states M*
and *q2* ∈ *FSM.states M*
and *ofsm-table-fix M* (λ*q* . *states M*) *0 q1* ≠ *ofsm-table-fix M* (λ*q* . *states M*) *0 q2*
shows (*find-first-distinct-ofsm-table M q1 q2*) = *Min* {*k* . *ofsm-table M* (λ*q* . *states M*) *k q1* ≠ *ofsm-table M* (λ*q* . *states M*) *k q2*
∧ (∀ *k′* . *k′* < *k* ⟶ *ofsm-table M* (λ*q* . *states M*) *k′ q1* = *ofsm-table M* (λ*q* . *states M*) *k′ q2*)}
(**is** *find-first-distinct-ofsm-table M q1 q2* = *Min ?ks*)
⟨*proof*⟩

lemma *find-first-distinct-ofsm-table′-is-first′* :
assumes *q1* ∈ *FSM.states M*
and *q2* ∈ *FSM.states M*
and *ofsm-table-fix M* (λ*q* . *states M*) *0 q1* ≠ *ofsm-table-fix M* (λ*q* . *states M*) *0 q2*
shows (*find-first-distinct-ofsm-table′ M q1 q2*) = *Min* {*k* . *ofsm-table M* (λ*q* . *states M*) *k q1* ≠ *ofsm-table M* (λ*q* . *states M*) *k q2*
∧ (∀ *k′* . *k′* < *k* ⟶ *ofsm-table M* (λ*q* . *states M*) *k′ q1* = *ofsm-table M* (λ*q* . *states M*) *k′ q2*)}
(**is** *find-first-distinct-ofsm-table′ M q1 q2* = *Min ?ks*)
and *find-first-distinct-ofsm-table′ M q1 q2* ≤ *size M* − *1*
⟨*proof*⟩

lemma *find-first-distinct-ofsm-table′-max* :
find-first-distinct-ofsm-table′ M q1 q2 ≤ *size M* − *1*
⟨*proof*⟩

lemma *find-first-distinct-ofsm-table-alt-def*:

find-first-distinct-ofsm-table M q1 q2 = find-first-distinct-ofsm-table′ M q1 q2
⟨*proof*⟩

13.3 Refining the Computation of Distinguishing Traces via OFSM Tables

fun *select-diverging-ofsm-table-io′* :: (*′a*::*linorder*,*′b*::*linorder*,*′c*::*linorder*) *fsm* ⇒ *′a* ⇒ *′a* ⇒ *nat* ⇒ (*′b* × *′c*) × (*′a option* × *′a option*) **where**
select-diverging-ofsm-table-io′ M q1 q2 k = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
ins = *inputs-as-list M*;
outs = *outputs-as-list M*;
table = *ofsm-lookup* (*the* (*Mapping.lookup tables* (*k*−*1*)));
f = (λ (*x*,*y*) . *case* (*h-obs M q1 x y*, *h-obs M q2 x y*)
of
(*Some q1′*, *Some q2′*) ⇒ *if table q1′* ≠ *table q2′*
then Some ((*x*,*y*),(*Some q1′*, *Some q2′*))
else None |
(*None*,*None*) ⇒ *None* |
(*Some q1′*, *None*) ⇒ *Some* ((*x*,*y*),(*Some q1′*, *None*)) |
(*None*, *Some q2′*) ⇒ *Some* ((*x*,*y*),(*None*, *Some q2′*)))
in
hd (*List.map-filter f* (*List.product ins outs*)))

lemma *select-diverging-ofsm-table-io-alt-def* :
assumes *k* ≤ *size M* − *1*
shows *select-diverging-ofsm-table-io M q1 q2 k* = *select-diverging-ofsm-table-io′ M q1 q2 k*
⟨*proof*⟩

fun *assemble-distinguishing-sequence-from-ofsm-table′* :: (*′a*::*linorder*,*′b*::*linorder*,*′c*::*linorder*) *fsm* ⇒ *′a* ⇒ *′a* ⇒ *nat* ⇒ (*′b* × *′c*) *list* **where**
assemble-distinguishing-sequence-from-ofsm-table′ M q1 q2 0 = [] |
assemble-distinguishing-sequence-from-ofsm-table′ M q1 q2 (*Suc k*) = (*case*
select-diverging-ofsm-table-io′ M q1 q2 (*Suc k*)
of
((*x*,*y*),(*Some q1′*,*Some q2′*)) ⇒ (*x*,*y*) # (*assemble-distinguishing-sequence-from-ofsm-table′ M q1′ q2′ k*) |
((*x*,*y*),-) ⇒ [(*x*,*y*)])

lemma *assemble-distinguishing-sequence-from-ofsm-table-alt-def* :
assumes *k* ≤ *size M* − *1*
shows *assemble-distinguishing-sequence-from-ofsm-table M q1 q2 k* = *assemble-distinguishing-sequence-from-ofsm-table′ M q1 q2 k*
⟨*proof*⟩

fun *get-distinguishing-sequence-from-ofsm-tables-refined* :: (*′a*::*linorder*,*′b*::*linorder*,*′c*::*linorder*) *fsm* ⇒ *′a* ⇒ *′a* ⇒ (*′b* × *′c*) *list* **where**
get-distinguishing-sequence-from-ofsm-tables-refined M q1 q2 = (*let*

k = *find-first-distinct-ofsm-table′ M q1 q2*
in assemble-distinguishing-sequence-from-ofsm-table′ M q1 q2 k)

lemma *get-distinguishing-sequence-from-ofsm-tables-refined-alt-def* :
get-distinguishing-sequence-from-ofsm-tables-refined M q1 q2 = *get-distinguishing-sequence-from-ofsm-tables M q1 q2*
⟨*proof*⟩

lemma *get-distinguishing-sequence-from-ofsm-tables-refined-distinguishes* :
assumes *observable M*
and *minimal M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *q1* ≠ *q2*
shows *distinguishes M q1 q2* (*get-distinguishing-sequence-from-ofsm-tables-refined M q1 q2*)
⟨*proof*⟩

```
fun select-diverging-ofsm-table-io-with-provided-tables :: (nat, ('a,'b,'c) ofsm-table)
mapping ⇒ ('a::linorder,'b::linorder,'c::linorder) fsm ⇒ 'a ⇒ 'a ⇒ nat ⇒ ('b ×
'c) × ('a option × 'a option) where
  select-diverging-ofsm-table-io-with-provided-tables tables M q1 q2 k = (let
      ins = inputs-as-list M;
      outs = outputs-as-list M;
      table = ofsm-lookup (the (Mapping.lookup tables (k−1)));
      f = (λ (x,y) . case (h-obs M q1 x y, h-obs M q2 x y)
                  of
                    (Some q1', Some q2') ⇒ if table q1' ≠ table q2'
                                           then Some ((x,y),(Some q1', Some q2'))
                                           else None |
                    (None,None)          ⇒ None |
                    (Some q1', None)     ⇒ Some ((x,y),(Some q1', None)) |
                    (None, Some q2')     ⇒ Some ((x,y),(None, Some q2')))
      in
        hd (List.map-filter f (List.product ins outs)))
```

lemma *select-diverging-ofsm-table-io-with-provided-tables-simp* :
select-diverging-ofsm-table-io-with-provided-tables (*compute-ofsm-tables M* (*size M* − *1*)) *M* = *select-diverging-ofsm-table-io′ M*
⟨*proof*⟩

```
fun assemble-distinguishing-sequence-from-ofsm-table-with-provided-tables :: (nat,
('a,'b,'c) ofsm-table) mapping ⇒ ('a::linorder,'b::linorder,'c::linorder) fsm ⇒ 'a
⇒ 'a ⇒ nat ⇒ ('b × 'c) list where
  assemble-distinguishing-sequence-from-ofsm-table-with-provided-tables tables M q1
q2 0 = [] |
  assemble-distinguishing-sequence-from-ofsm-table-with-provided-tables tables M q1
```

q2 (*Suc k*) = (*case*
select-diverging-ofsm-table-io-with-provided-tables tables M q1 q2 (*Suc k*)
of
((*x*,*y*),(*Some q1'*,*Some q2'*)) ⇒ (*x*,*y*) # (*assemble-distinguishing-sequence-from-ofsm-table-with-provided-tables tables M q1' q2' k*) |
((*x*,*y*),-) ⇒ [(*x*,*y*)])

lemma *assemble-distinguishing-sequence-from-ofsm-table-with-provided-tables-simp*
:
assemble-distinguishing-sequence-from-ofsm-table-with-provided-tables (*compute-ofsm-tables M* (*size M* − *1*)) *M q1 q2 k*= *assemble-distinguishing-sequence-from-ofsm-table' M q1 q2 k*
⟨*proof*⟩

lemma *get-distinguishing-sequence-from-ofsm-tables-refined-code*[*code*] :
get-distinguishing-sequence-from-ofsm-tables-refined M q1 q2 = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
k = (*if* (*q1* ∈ *states M*
∧ *q2* ∈ *states M*
∧ (*ofsm-lookup* (*the* (*Mapping.lookup tables* (*size M* − *1*))) *q1*
≠ *ofsm-lookup* (*the* (*Mapping.lookup tables* (*size M* − *1*))) *q2*))
then the (*find-index* (λ *i* . *ofsm-lookup* (*the* (*Mapping.lookup tables i*)) *q1* ≠ *ofsm-lookup* (*the* (*Mapping.lookup tables i*)) *q2*) [*0*..<*size M*])
else 0)
in assemble-distinguishing-sequence-from-ofsm-table-with-provided-tables tables M q1 q2 k)
⟨*proof*⟩

fun *get-distinguishing-sequence-from-ofsm-tables-with-provided-tables* :: (*nat*, (*'a*,*'b*,*'c*) *ofsm-table*) *mapping* ⇒ (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* ⇒ *'a* ⇒ *'a* ⇒ (*'b* × *'c*) *list* **where**
get-distinguishing-sequence-from-ofsm-tables-with-provided-tables tables M q1 q2 = (*let*
k = (*if* (*q1* ∈ *states M*
∧ *q2* ∈ *states M*
∧ (*ofsm-lookup* (*the* (*Mapping.lookup tables* (*size M* − *1*))) *q1*
≠ *ofsm-lookup* (*the* (*Mapping.lookup tables* (*size M* − *1*))) *q2*))
then the (*find-index* (λ *i* . *ofsm-lookup* (*the* (*Mapping.lookup tables i*)) *q1* ≠ *ofsm-lookup* (*the* (*Mapping.lookup tables i*)) *q2*) [*0*..<*size M*])
else 0)
in assemble-distinguishing-sequence-from-ofsm-table-with-provided-tables tables M q1 q2 k)

lemma *get-distinguishing-sequence-from-ofsm-tables-with-provided-tables-simp* :
get-distinguishing-sequence-from-ofsm-tables-with-provided-tables (*compute-ofsm-tables M* (*size M* − *1*)) *M* = *get-distinguishing-sequence-from-ofsm-tables-refined M*
⟨*proof*⟩

lemma *get-distinguishing-sequence-from-ofsm-tables-precomputed*:
get-distinguishing-sequence-from-ofsm-tables M = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
distMap = *mapping-of* (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provided-tables M q1 q2*))
(*filter* (λ *qq* . *fst qq* ≠ *snd qq*) (*List.product* (*states-as-list M*) (*states-as-list M*))));
distHelper = (λ *q1 q2* . *if q1* ∈ *states M* ∧ *q2* ∈ *states M* ∧ *q1* ≠ *q2 then the* (*Mapping.lookup distMap* (*q1*,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables M q1 q2*)
in distHelper)
⟨*proof*⟩

lemma *get-distinguishing-sequence-from-ofsm-tables-with-provided-tables-distinguishes*
:
assumes *observable M*
and *minimal M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *q1* ≠ *q2*
shows *distinguishes M q1 q2* (*get-distinguishing-sequence-from-ofsm-tables-with-provided-tables* (*compute-ofsm-tables M* (*size M* − *1*)) *M q1 q2*)
⟨*proof*⟩

13.4 Refining Minimisation

fun *minimise-refined* :: ('*a* :: *linorder*,'*b* :: *linorder*,'*c* :: *linorder*) *fsm* ⇒ ('*a set*,'*b*,'*c*) *fsm* **where**
minimise-refined M = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
eq-class = (*ofsm-lookup* (*the* (*Mapping.lookup tables* (*size M* − *1*))));
ts = (λ *t* . (*eq-class* (*t-source t*), *t-input t*, *t-output t*, *eq-class* (*t-target t*))) ‘ (*transitions M*);
q0 = *eq-class* (*initial M*);
eq-states = *eq-class* |‘| *fstates M*;
M' = *create-unconnected-fsm-from-fsets q0 eq-states* (*finputs M*) (*foutputs M*)
in add-transitions M' ts)

lemma *minimise-refined-is-minimise*[*code*] : *minimise M* = *minimise-refined M*
⟨*proof*⟩

end

14 Transformation to Language-Equivalent Prime FSMs

This theory describes the transformation of FSMs into language-equivalent FSMs that are prime, that is: observable, minimal and initially connected.

theory *Prime-Transformation*
imports *Minimisation Observability State-Cover OFSM-Tables-Refined HOL−Library.List-Lexorder Native-Word.Uint64*
begin

14.1 Helper Functions

The following functions transform FSMs whose states are Sets or FSets into language-equivalent fsms whose states are lists. These steps are required in the chosen implementation of the transformation function, as Sets or FSets are not instances of linorder.

lemma *linorder-fset-list-bij* : *bij-betw sorted-list-of-fset xs* (*sorted-list-of-fset* ‘ *xs*)
⟨*proof*⟩

lemma *linorder-set-list-bij* :
assumes $\bigwedge x . x \in xs \Longrightarrow$ *finite x*
shows *bij-betw sorted-list-of-set xs* (*sorted-list-of-set* ‘ *xs*)
⟨*proof*⟩

definition *fset-states-to-list-states* :: (('*a*::*linorder*) *fset*,'*b*,'*c*) *fsm* ⇒ ('*a list*,'*b*,'*c*) *fsm* **where**
fset-states-to-list-states M = *rename-states M sorted-list-of-fset*

definition *set-states-to-list-states* :: (('*a*::*linorder*) *set*,'*b*,'*c*) *fsm* ⇒ ('*a list*,'*b*,'*c*) *fsm* **where**
set-states-to-list-states M = *rename-states M sorted-list-of-set*

lemma *fset-states-to-list-states-language* :
L (*fset-states-to-list-states M*) = *L M*
⟨*proof*⟩

lemma *set-states-to-list-states-language* :
assumes $\bigwedge x . x \in$ *states M* $\Longrightarrow$ *finite x*
shows *L* (*set-states-to-list-states M*) = *L M*
⟨*proof*⟩

lemma *fset-states-to-list-states-observable* :
assumes *observable M*
shows *observable* (*fset-states-to-list-states M*)
⟨*proof*⟩

lemma *set-states-to-list-states-observable* :

assumes $\bigwedge x . x \in$ *states M* $\Longrightarrow$ *finite x*
assumes *observable M*
shows *observable* (*set-states-to-list-states M*)
⟨*proof*⟩

lemma *fset-states-to-list-states-minimal* :
assumes *minimal M*
shows *minimal* (*fset-states-to-list-states M*)
⟨*proof*⟩

lemma *set-states-to-list-states-minimal* :
assumes $\bigwedge x . x \in$ *states M* $\Longrightarrow$ *finite x*
assumes *minimal M*
shows *minimal* (*set-states-to-list-states M*)
⟨*proof*⟩

14.2 The Transformation Algorithm

definition *to-prime* :: (*'a* :: *linorder*,*'b* :: *linorder*,*'c* :: *linorder*) *fsm* ⇒ (*integer*,*'b*,*'c*) *fsm* **where**

to-prime M = *restrict-to-reachable-states* (
  index-states-integer (
    set-states-to-list-states (
      minimise-refined (
        index-states (
          fset-states-to-list-states (
            make-observable (
              restrict-to-reachable-states M)))))))

lemma *to-prime-props* :
L (*to-prime M*) = *L M*
observable (*to-prime M*)
minimal (*to-prime M*)
reachable-states (*to-prime M*) = *states* (*to-prime M*)
inputs (*to-prime M*) = *inputs M*
outputs (*to-prime M*) = *outputs M*
⟨*proof*⟩

14.3 Renaming states to Words

lemma *uint64-nat-bij* : (x :: *nat*) < *2^64* $\Longrightarrow$ *nat-of-uint64* (*uint64-of-nat x*) = *x*
⟨*proof*⟩

fun *index-states-uint64* :: (*'a*::*linorder*,*'b*,*'c*) *fsm* ⇒ (*uint64*,*'b*,*'c*) *fsm* **where**
index-states-uint64 M = *rename-states M* (*uint64-of-nat* ∘ *assign-indices* (*states M*))

lemma *assign-indices-uint64-bij-betw* :
assumes *size M* < *2^64*

shows *bij-betw* (*uint64-of-nat* ∘ *assign-indices* (*states M*)) (*FSM.states M*) ((*uint64-of-nat* ∘ *assign-indices* (*states M*)) ‘ *FSM.states M*)
⟨*proof*⟩

lemma *index-states-uint64-language* :
assumes *size M* < *2^64*
shows *L* (*index-states-uint64 M*) = *L M*
⟨*proof*⟩

lemma *index-states-uint64-observable* :
assumes *size M* < *2^64* **and** *observable M*
shows *observable* (*index-states-uint64 M*)
⟨*proof*⟩

lemma *index-states-uint64-minimal* :
assumes *size M* < *2^64* **and** *minimal M*
shows *minimal* (*index-states-uint64 M*)
⟨*proof*⟩

definition *to-prime-uint64* :: (*'a* :: *linorder*,*'b* :: *linorder*,*'c* :: *linorder*) *fsm* ⇒ (*uint64*,*'b*,*'c*) *fsm* **where**
to-prime-uint64 M = *restrict-to-reachable-states* (*index-states-uint64* (*to-prime M*))

lemma *to-prime-uint64-props* :
assumes *size* (*to-prime M*) < *2^64*
shows
L (*to-prime-uint64 M*) = *L M*
observable (*to-prime-uint64 M*)
minimal (*to-prime-uint64 M*)
reachable-states (*to-prime-uint64 M*) = *states* (*to-prime-uint64 M*)
inputs (*to-prime-uint64 M*) = *inputs M*
outputs (*to-prime-uint64 M*) = *outputs M*
⟨*proof*⟩

end

15 Convergence of Traces

This theory defines convergence of traces in observable FSMs and provides results on sufficient conditions to establish that two traces converge. Furthermore it is shown how convergence can be employed in proving language equivalence.

theory *Convergence*
imports ../*Minimisation* ../*Distinguishability* ../*State-Cover HOL−Library.List-Lexorder*

begin

15.1 Basic Definitions

fun *converge* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $('b \times 'c)$ *list* $\Rightarrow$ $('b \times 'c)$ *list* $\Rightarrow$ *bool* **where**
converge $M\ \pi\ \tau = (\pi \in L\ M \wedge \tau \in L\ M \wedge (LS\ M\ (\textit{after-initial}\ M\ \pi) = LS\ M\ (\textit{after-initial}\ M\ \tau)))$

fun *preserves-divergence* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $('d,'b,'c)$ *fsm* $\Rightarrow$ $('b \times 'c)$ *list set* $\Rightarrow$ *bool* **where**
preserves-divergence $M1\ M2\ A = (\forall\ \alpha \in L\ M1 \cap A\ .\ \forall\ \beta \in L\ M1 \cap A\ .\ \neg\ \textit{converge}\ M1\ \alpha\ \beta \longrightarrow \neg\ \textit{converge}\ M2\ \alpha\ \beta)$

fun *preserves-convergence* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $('d,'b,'c)$ *fsm* $\Rightarrow$ $('b \times 'c)$ *list set* $\Rightarrow$ *bool* **where**
preserves-convergence $M1\ M2\ A = (\forall\ \alpha \in L\ M1 \cap A\ .\ \forall\ \beta \in L\ M1 \cap A\ .\ \textit{converge}\ M1\ \alpha\ \beta \longrightarrow \textit{converge}\ M2\ \alpha\ \beta)$

lemma *converge-refl* :
assumes $\alpha \in L\ M$
shows *converge* $M\ \alpha\ \alpha$
⟨*proof*⟩

lemma *convergence-minimal* :
assumes *minimal M*
and *observable M*
and $\alpha \in L\ M$
and $\beta \in L\ M$
shows *converge* $M\ \alpha\ \beta = ((\textit{after-initial}\ M\ \alpha) = (\textit{after-initial}\ M\ \beta))$
⟨*proof*⟩

lemma *state-cover-assignment-diverges* :
assumes *observable M*
and *minimal M*
and *is-state-cover-assignment M f*
and $q1 \in \textit{reachable-states}\ M$
and $q2 \in \textit{reachable-states}\ M$
and $q1 \neq q2$
shows $\neg$ *converge* $M\ (f\ q1)\ (f\ q2)$
⟨*proof*⟩

lemma *converge-extend* :
assumes *observable M*
and *converge* $M\ \alpha\ \beta$
and $\alpha @ \gamma \in L\ M$
and $\beta \in L\ M$
shows $\beta @ \gamma \in L\ M$

⟨*proof*⟩

lemma *converge-append* :
assumes *observable M*
and *converge M* α β
and $\alpha @ \gamma \in L\ M$
and $\beta \in L\ M$
shows *converge M* $(\alpha @ \gamma)$ $(\beta @ \gamma)$
⟨*proof*⟩

lemma *non-initialized-state-cover-assignment-diverges* :
assumes *observable M*
and *minimal M*
and $\bigwedge q\ .\ q \in$ *reachable-states M* $\Longrightarrow q \in$ *io-targets M* $(f\ q)$ (*initial M*)
and $\bigwedge q\ .\ q \in$ *reachable-states M* $\Longrightarrow f\ q \in L\ M \cap\ SC$
and $q1 \in$ *reachable-states M*
and $q2 \in$ *reachable-states M*
and $q1 \neq q2$
shows $\neg$ *converge M* $(f\ q1)$ $(f\ q2)$
⟨*proof*⟩

lemma *converge-trans-2* :
assumes *observable M* **and** *minimal M* **and** *converge M u v*
shows *converge M* $(u@w1)$ $(u@w2)$ = *converge M* $(v@w1)$ $(v@w2)$
converge M $(u@w1)$ $(u@w2)$ = *converge M* $(u@w1)$ $(v@w2)$
converge M $(u@w1)$ $(u@w2)$ = *converge M* $(v@w1)$ $(u@w2)$
⟨*proof*⟩

lemma *preserves-divergence-converge-insert* :
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *converge M1 u v*
and *converge M2 u v*
and *preserves-divergence M1 M2 X*
and $u \in X$
shows *preserves-divergence M1 M2* (*Set.insert v X*)
⟨*proof*⟩

lemma *preserves-divergence-converge-replace* :
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*

and *converge M1 u v*
and *converge M2 u v*
and *preserves-divergence M1 M2* (*Set.insert u X*)
shows *preserves-divergence M1 M2* (*Set.insert v X*)
⟨*proof*⟩

lemma *preserves-divergence-converge-replace-iff* :
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *converge M1 u v*
and *converge M2 u v*
shows *preserves-divergence M1 M2* (*Set.insert u X*) = *preserves-divergence M1 M2* (*Set.insert v X*)
⟨*proof*⟩

lemma *preserves-divergence-subset* :
assumes *preserves-divergence M1 M2 B*
and $A \subseteq B$
shows *preserves-divergence M1 M2 A*
⟨*proof*⟩

lemma *preserves-divergence-insertI* :
assumes *preserves-divergence M1 M2 X*
and $\bigwedge \alpha\ .\ \alpha \in L\ M1 \cap X \Longrightarrow \beta \in L\ M1 \Longrightarrow \neg converge\ M1\ \alpha\ \beta \Longrightarrow \neg converge\ M2\ \alpha\ \beta$
shows *preserves-divergence M1 M2* (*Set.insert* β *X*)
⟨*proof*⟩

lemma *preserves-divergence-insertE* :
assumes *preserves-divergence M1 M2* (*Set.insert* β *X*)
shows *preserves-divergence M1 M2 X*
and $\bigwedge \alpha\ .\ \alpha \in L\ M1 \cap X \Longrightarrow \beta \in L\ M1 \Longrightarrow \neg converge\ M1\ \alpha\ \beta \Longrightarrow \neg converge\ M2\ \alpha\ \beta$
⟨*proof*⟩

lemma *distinguishes-diverge-prefix* :
assumes *observable M*
and *distinguishes M* (*after-initial M u*) (*after-initial M v*) *w*
and $u \in L\ M$
and $v \in L\ M$
and $w' \in set\ (prefixes\ w)$
and $w' \in LS\ M\ (after\text{-}initial\ M\ u)$
and $w' \in LS\ M\ (after\text{-}initial\ M\ v)$
shows $\neg converge\ M\ (u@w')\ (v@w')$
⟨*proof*⟩

lemma *converge-distinguishable-helper* :

assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *converge M1* π α
and *converge M2* π α
and *converge M1* τ β
and *converge M2* τ β
and *distinguishes M2* (*after-initial M2* π) (*after-initial M2* τ) v
and L *M1* $\cap$ $\{\alpha@v,\beta@v\}$ = L *M2* $\cap$ $\{\alpha@v,\beta@v\}$
shows (*after-initial M1* π) $\neq$ (*after-initial M1* τ)
⟨*proof*⟩

lemma *converge-append-language-iff* :
assumes *observable M*
and *converge M* α β
shows ($\alpha@\gamma \in L$ *M*) = ($\beta@\gamma \in L$ *M*)
⟨*proof*⟩

lemma *converge-append-iff* :
assumes *observable M*
and *converge M* α β
shows *converge M* γ ($\alpha@\omega$) = *converge M* γ ($\beta@\omega$)
⟨*proof*⟩

lemma *after-distinguishes-language* :
assumes *observable M1*
and $\alpha \in L$ *M1*
and $\beta \in L$ *M1*
and *distinguishes M1* (*after-initial M1* α) (*after-initial M1* β) γ
shows ($\alpha@\gamma \in L$ *M1*) $\neq$ ($\beta@\gamma \in L$ *M1*)
⟨*proof*⟩

lemma *distinguish-diverge* :
assumes *observable M1*
and *observable M2*
and *distinguishes M1* (*after-initial M1 u*) (*after-initial M1 v*) γ
and u @ $\gamma \in T$
and v @ $\gamma \in T$
and $u \in L$ *M1*
and $v \in L$ *M1*
and L *M1* $\cap$ T = L *M2* $\cap$ T
shows $\neg$ *converge M2 u v*
⟨*proof*⟩

lemma *distinguish-converge-diverge* :
assumes *observable M1*

and *observable M2*
and *minimal M1*
and $u' \in L$ *M1*
and $v' \in L$ *M1*
and *converge M1 u u'*
and *converge M1 v v'*
and *converge M2 u u'*
and *converge M2 v v'*
and *distinguishes M1* (*after-initial M1 u*) (*after-initial M1 v*) γ
and u' @ $\gamma \in T$
and v' @ $\gamma \in T$
and L *M1* $\cap T = L$ *M2* $\cap T$
shows ¬ *converge M2 u v*
⟨*proof*⟩

lemma *diverge-prefix* :
assumes *observable M*
and α@$\gamma \in L$ *M*
and β@$\gamma \in L$ *M*
and ¬ *converge M* (α@γ) (β@γ)
shows ¬ *converge M* α β
⟨*proof*⟩

lemma *converge-sym*: *converge M u v* = *converge M v u*
⟨*proof*⟩

lemma *state-cover-transition-converges* :
assumes *observable M*
and *is-state-cover-assignment M V*
and $t \in$ *transitions M*
and *t-source t* $\in$ *reachable-states M*
shows *converge M* ((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]) (*V* (*t-target t*))
⟨*proof*⟩

lemma *equivalence-preserves-divergence* :
assumes *observable M*
and *observable I*
and L $M = L$ I
shows *preserves-divergence M I A*
⟨*proof*⟩

15.2 Sufficient Conditions for Convergence

The following lemma provides a condition for convergence that assumes the existence of a single state cover covering all extensions of length up to (m - |M1|). This is too restrictive for the SPYH method but could be used in the SPY method. The proof idea has been developed by Wen-ling Huang and adapted by the author to avoid requiring the SC to cover traces that contain a proper prefix already not in the language of FSM M1.

lemma *sufficient-condition-for-convergence-in-SPY-method* :
fixes *M1* :: ('*a*,'*b*,'*c*) *fsm*
fixes *M2* :: ('*d*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and $size\text{-}r\ M1 \leq m$
and $size\ M2 \leq m$
and $L\ M1 \cap T = L\ M2 \cap T$
and $\pi \in L\ M1 \cap T$
and $\tau \in L\ M1 \cap T$
and $converge\ M1\ \pi\ \tau$
and $SC \subseteq T$
and $\bigwedge q\ .\ q \in reachable\text{-}states\ M1 \Longrightarrow \exists\ io \in L\ M1 \cap SC\ .\ q \in io\text{-}targets\ M1\ io\ (initial\ M1)$
and *preserves-divergence M1 M2 SC*
and $\bigwedge \gamma\ x\ y\ .\ length\ \gamma < m - size\text{-}r\ M1 \Longrightarrow$
$\gamma \in LS\ M1\ (after\text{-}initial\ M1\ \pi) \Longrightarrow$
$x \in inputs\ M1 \Longrightarrow$
$y \in outputs\ M1 \Longrightarrow$
$\exists\ \alpha\ \beta\ .\ converge\ M1\ \alpha\ (\pi@\gamma) \wedge$
$converge\ M2\ \alpha\ (\pi@\gamma) \wedge$
$converge\ M1\ \beta\ (\tau@\gamma) \wedge$
$converge\ M2\ \beta\ (\tau@\gamma) \wedge$
$\alpha \in SC \wedge$
$\alpha@[(x,y)] \in SC \wedge$
$\beta \in SC \wedge$
$\beta@[(x,y)] \in SC$
and $\exists\ \alpha\ \beta\ .\ converge\ M1\ \alpha\ \pi \wedge$
$converge\ M2\ \alpha\ \pi \wedge$
$converge\ M1\ \beta\ \tau \wedge$
$converge\ M2\ \beta\ \tau \wedge$
$\alpha \in SC \wedge$
$\beta \in SC$
and $inputs\ M2 = inputs\ M1$
and $outputs\ M2 = outputs\ M1$
shows $converge\ M2\ \pi\ \tau$
⟨*proof*⟩

lemma *preserves-divergence-minimally-distinguishing-prefixes-lower-bound* :
fixes *M1* :: ('*a*,'*b*,'*c*) *fsm*
fixes *M2* :: ('*d*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*

and *minimal M2*
and *converge M1 u v*
and ¬*converge M2 u v*
and $u \in L$ *M2*
and $v \in L$ *M2*
and *minimally-distinguishes M2* (*after-initial M2 u*) (*after-initial M2 v*) *w*
and *wp* ∈ *list.set* (*prefixes w*)
and *wp* ≠ *w*
and *wp* ∈ *LS M1* (*after-initial M1 u*) ∩ *LS M1* (*after-initial M1 v*)
and *preserves-divergence M1 M2* {α@γ | α γ . α ∈ {*u*,*v*} ∧ γ ∈ *list.set* (*prefixes wp*)}
and *L M1* ∩ {α@γ | α γ . α ∈ {*u*,*v*} ∧ γ ∈ *list.set* (*prefixes wp*)} = *L M2* ∩ {α@γ | α γ . α ∈ {*u*,*v*} ∧ γ ∈ *list.set* (*prefixes wp*)}
shows *card* (*after-initial M2* ‘ {α@γ | α γ . α ∈ {*u*,*v*} ∧ γ ∈ *list.set* (*prefixes wp*)}) ≥ *length wp* + (*card* (*FSM.after M1* (*after-initial M1 u*) ‘ (*list.set* (*prefixes wp*)))) + *1*
⟨*proof*⟩

lemma *sufficient-condition-for-convergence* :
fixes *M1* :: (′*a*,′*b*,′*c*) *fsm*
fixes *M2* :: (′*d*,′*b*,′*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
and *converge M1* π τ
and *L M1* ∩ *T* = *L M2* ∩ *T*
and ⋀ γ *x y* . *length* (γ@[(*x*,*y*)]) ≤ *m* − *size-r M1* ⟹
γ ∈ *LS M1* (*after-initial M1* π) ⟹
x ∈ *inputs M1* ⟹ *y* ∈ *outputs M1* ⟹
∃ *SC* α β . *SC* ⊆ *T*
∧ *is-state-cover M1 SC*
∧ {ω@ω′ | ω ω′ . ω ∈ {α,β} ∧ ω′ ∈ *list.set* (*prefixes* (γ@[(*x*,*y*)]))} ⊆ *SC*
∧ *converge M1* π α
∧ *converge M2* π α
∧ *converge M1* τ β
∧ *converge M2* τ β
∧ *preserves-divergence M1 M2 SC*
and ∃ *SC* α β . *SC* ⊆ *T*
∧ *is-state-cover M1 SC*
∧ α ∈ *SC* ∧ β ∈ *SC*
∧ *converge M1* π α
∧ *converge M2* π α

$\land$ *converge M1* τ β
$\land$ *converge M2* τ β
$\land$ *preserves-divergence M1 M2 SC*
shows *converge M2* π τ
⟨*proof*⟩

lemma *establish-convergence-from-pass* :
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* $\leq m$
and *size M2* $\leq m$
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
and *is-state-cover-assignment M1 V*
and *L M1* $\cap$ (*V* ‘ *reachable-states M1*) = *L M2* $\cap$ *V* ‘ *reachable-states M1*
and *converge M1 u v*
and $u \in$ *L M2*
and $v \in$ *L M2*
and *prop1*: $\bigwedge\gamma\ x\ y.$
length (γ @ [(*x*, *y*)]) $\leq$ (*m* $-$ *size-r M1*) $\Longrightarrow$
$\gamma \in$ *LS M1* (*after-initial M1 u*) $\Longrightarrow$
$x \in$ *FSM.inputs M1* $\Longrightarrow$
$y \in$ *FSM.outputs M1* $\Longrightarrow$
L M1 $\cap$ ((*V* ‘ *reachable-states M1*) $\cup$ {ω @ ω' |ω ω'. $\omega \in$ {*u*, *v*} $\land$ $\omega' \in$ *list.set* (*prefixes* (γ @ [(*x*, *y*)]))}) =
L M2 $\cap$ ((*V* ‘ *reachable-states M1*) $\cup$ {ω @ ω' |ω ω'. $\omega \in$ {*u*, *v*} $\land$ $\omega' \in$ *list.set* (*prefixes* (γ @ [(*x*, *y*)]))}) $\land$
preserves-divergence M1 M2 ((*V* ‘ *reachable-states M1*) $\cup$ {ω @ ω' |ω ω'. $\omega \in$ {*u*, *v*} $\land$ $\omega' \in$ *list.set* (*prefixes* (γ @ [(*x*, *y*)]))})
and *prop2*: *preserves-divergence M1 M2* ((*V* ‘ *reachable-states M1*) $\cup$ {*u*, *v*})
shows *converge M2 u v*
⟨*proof*⟩

15.3 Proving Language Equivalence by Establishing a Convergence Preserving Initialised Transition Cover

definition *transition-cover* :: ('*a*,'*b*,'*c*) *fsm* $\Rightarrow$ ('*b* $\times$ '*c*) *list set* $\Rightarrow$ *bool* **where**
transition-cover M A = ($\forall$ *q* $\in$ *reachable-states M* . $\forall$ *x* $\in$ *inputs M* . $\forall$ *y* $\in$ *outputs M* . $\exists$ α. $\alpha \in A \land \alpha$@[(*x*,*y*)] $\in A \land \alpha \in$ *L M* $\land$ *after-initial M* α = *q*)

lemma *initialised-convergence-preserving-transition-cover-is-complete* :
fixes *M1* :: ('*a*,'*b*,'*c*) *fsm*
fixes *M2* :: ('*d*,'*b*,'*c*) *fsm*
assumes *observable M1*

and *observable M2*
and *minimal M1*
and *minimal M2*
and *inputs M2 = inputs M1*
and *outputs M2 = outputs M1*
and *L M1 ∩ T = L M2 ∩ T*
and *A ⊆ T*
and *transition-cover M1 A*
and *[] ∈ A*
and *preserves-convergence M1 M2 A*
shows *L M1 = L M2*
⟨*proof*⟩

end

16 Convergence Graphs

This theory introduces the invariants required for the initialisation, insertion, lookup, and merge operations on convergence graphs.

theory *Convergence-Graph*
imports *Convergence ../Prefix-Tree*
begin

lemma *after-distinguishes-diverge* :
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *α ∈ L M1*
and *β ∈ L M1*
and *γ ∈ set (after T1 α) ∩ set (after T1 β)*
and *distinguishes M1 (after-initial M1 α) (after-initial M1 β) γ*
and *L M1 ∩ set T1 = L M2 ∩ set T1*
shows *¬converge M2 α β*
⟨*proof*⟩

16.1 Required Invariants on Convergence Graphs

definition *convergence-graph-lookup-invar* :: *('a,'b,'c) fsm ⇒ ('e,'b,'c) fsm ⇒ ('d ⇒ ('b×'c) list ⇒ ('b×'c) list list) ⇒ 'd ⇒ bool*
where
convergence-graph-lookup-invar M1 M2 cg-lookup G = (∀ α . α ∈ L M1 ⟶ α ∈ L M2 ⟶ α ∈ list.set (cg-lookup G α) ∧ (∀ β . β ∈ list.set (cg-lookup G α) ⟶ converge M1 α β ∧ converge M2 α β))

lemma *convergence-graph-lookup-invar-simp*:
  assumes *convergence-graph-lookup-invar M1 M2 cg-lookup G*
  and $\alpha \in L\ M1$ **and** $\alpha \in L\ M2$
  and $\beta \in list.set\ (cg\text{-}lookup\ G\ \alpha)$
shows *converge M1* α β **and** *converge M2* α β
  ⟨*proof*⟩

definition *convergence-graph-initial-invar* :: $('a,'b,'c)$ *fsm* ⇒ $('e,'b,'c)$ *fsm* ⇒
  $('d \Rightarrow ('b\times'c)\ list \Rightarrow ('b\times'c)\ list\ list) \Rightarrow$
  $(('a,'b,'c)\ fsm \Rightarrow ('b\times'c)\ prefix\text{-}tree \Rightarrow 'd) \Rightarrow$
  bool
  where
  convergence-graph-initial-invar M1 M2 cg-lookup cg-initial = $(\forall\ T\ .\ (L\ M1 \cap set\ T = (L\ M2 \cap set\ T)) \longrightarrow finite\text{-}tree\ T \longrightarrow$ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*cg-initial M1 T*))

definition *convergence-graph-insert-invar* :: $('a,'b,'c)$ *fsm* ⇒ $('e,'b,'c)$ *fsm* ⇒
  $('d \Rightarrow ('b\times'c)\ list \Rightarrow ('b\times'c)\ list\ list) \Rightarrow$
  $('d \Rightarrow ('b\times'c)\ list \Rightarrow 'd) \Rightarrow$
  bool
  where
  convergence-graph-insert-invar M1 M2 cg-lookup cg-insert = $(\forall\ G\ \gamma\ .\ \gamma \in L\ M1 \longrightarrow \gamma \in L\ M2 \longrightarrow$ *convergence-graph-lookup-invar M1 M2 cg-lookup G* ⟶ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*cg-insert G* γ))

definition *convergence-graph-merge-invar* :: $('a,'b,'c)$ *fsm* ⇒ $('e,'b,'c)$ *fsm* ⇒
  $('d \Rightarrow ('b\times'c)\ list \Rightarrow ('b\times'c)\ list\ list) \Rightarrow$
  $('d \Rightarrow ('b\times'c)\ list \Rightarrow ('b\times'c)\ list \Rightarrow 'd) \Rightarrow$
  bool
  where
  convergence-graph-merge-invar M1 M2 cg-lookup cg-merge = $(\forall\ G\ \gamma\ \gamma'.$ *converge M1* $\gamma\ \gamma' \longrightarrow$ *converge M2* $\gamma\ \gamma' \longrightarrow$ *convergence-graph-lookup-invar M1 M2 cg-lookup G* ⟶ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*cg-merge G* $\gamma\ \gamma'$))

end

17 An Always-Empty Convergence Graph

This theory implements a convergence graph that always returns an empty list for any lookup. By using this graph it is possible to represent methods via the SPY and H-Frameworks that do not distribute distinguishing traces over converging traces.

theory *Empty-Convergence-Graph*
imports *Convergence-Graph*
begin

type-synonym *empty-cg* = *unit*

definition *empty-cg-empty* :: *empty-cg* **where**
empty-cg-empty = ()

definition *empty-cg-initial* :: ((*'a*,*'b*,*'c*) *fsm* ⇒ (*'b*×*'c*) *prefix-tree* ⇒ *empty-cg*)
where
empty-cg-initial M T = *empty-cg-empty*

definition *empty-cg-insert* :: (*empty-cg* ⇒ (*'b*×*'c*) *list* ⇒ *empty-cg*) **where**
empty-cg-insert G v = *empty-cg-empty*

definition *empty-cg-lookup* :: (*empty-cg* ⇒ (*'b*×*'c*) *list* ⇒ (*'b*×*'c*) *list list*) **where**
empty-cg-lookup G v = [*v*]

definition *empty-cg-merge* :: (*empty-cg* ⇒ (*'b*×*'c*) *list* ⇒ (*'b*×*'c*) *list* ⇒ *empty-cg*)
where
empty-cg-merge G u v = *empty-cg-empty*

lemma *empty-graph-initial-invar*: *convergence-graph-initial-invar M1 M2 empty-cg-lookup empty-cg-initial*
⟨*proof*⟩

lemma *empty-graph-insert-invar*: *convergence-graph-insert-invar M1 M2 empty-cg-lookup empty-cg-insert*
⟨*proof*⟩

lemma *empty-graph-merge-invar*: *convergence-graph-merge-invar M1 M2 empty-cg-lookup empty-cg-merge*
⟨*proof*⟩

end

18 H-Framework

This theory defines the H-Framework and provides completeness properties.

theory *H-Framework*
imports *Convergence-Graph ../Prefix-Tree ../State-Cover*
begin

18.1 Abstract H-Condition

definition *satisfies-abstract-h-condition* :: (*'a*,*'b*,*'c*) *fsm* ⇒ (*'e*,*'b*,*'c*) *fsm* ⇒ (*'a*,*'b*,*'c*) *state-cover-assignment* ⇒ *nat* ⇒ *bool* **where**
satisfies-abstract-h-condition M1 M2 V m = (∀ *q* γ .
q ∈ *reachable-states M1* ⟶
length γ ≤ *Suc* (*m*−*size-r M1*) ⟶

list.set $\gamma \subseteq$ *inputs M1* $\times$ *outputs M1* $\longrightarrow$
butlast $\gamma \in$ *LS M1 q* $\longrightarrow$
(*let traces* = (*V* ‘ *reachable-states M1*)
$\cup$ {*V q* @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)}
in (*L M1* $\cap$ *traces* = *L M2* $\cap$ *traces*)
$\wedge$ *preserves-divergence M1 M2 traces*))

lemma *abstract-h-condition-exhaustiveness* :
assumes *observable M*
and *observable I*
and *minimal M*
and *size I* $\leq$ *m*
and *m* $\geq$ *size-r M*
and *inputs I* = *inputs M*
and *outputs I* = *outputs M*
and *is-state-cover-assignment M V*
and *satisfies-abstract-h-condition M I V m*
shows *L M* = *L I*
⟨*proof*⟩

lemma *abstract-h-condition-soundness* :
assumes *observable M*
and *observable I*
and *is-state-cover-assignment M V*
and *L M* = *L I*
shows *satisfies-abstract-h-condition M I V m*
⟨*proof*⟩

lemma *abstract-h-condition-completeness* :
assumes *observable M*
and *observable I*
and *minimal M*
and *size I* $\leq$ *m*
and *m* $\geq$ *size-r M*
and *inputs I* = *inputs M*
and *outputs I* = *outputs M*
and *is-state-cover-assignment M V*
shows *satisfies-abstract-h-condition M I V m* $\longleftrightarrow$ (*L M* = *L I*)
⟨*proof*⟩

18.2 Definition of the Framework

definition *h-framework* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* $\Rightarrow$

$((\prime a,\prime b,\prime c)$ *fsm* $\Rightarrow$ $(\prime a,\prime b,\prime c)$ *state-cover-assignment*$) \Rightarrow$

$((\prime a,\prime b,\prime c)$ *fsm* $\Rightarrow$ $(\prime a,\prime b,\prime c)$ *state-cover-assignment* $\Rightarrow$ $((\prime a,\prime b,\prime c)$ *fsm* $\Rightarrow$ $(\prime b\times\prime c)$ *prefix-tree* $\Rightarrow \prime d) \Rightarrow (\prime d \Rightarrow (\prime b\times\prime c)$ *list* $\Rightarrow \prime d) \Rightarrow (\prime d \Rightarrow (\prime b\times\prime c)$ *list* $\Rightarrow (\prime b\times\prime c)$ *list list*$) \Rightarrow ((\prime b\times\prime c)$ *prefix-tree* $\times \prime d)) \Rightarrow$
$((\prime a,\prime b,\prime c)$ *fsm* $\Rightarrow$ $(\prime a,\prime b,\prime c)$ *state-cover-assignment* $\Rightarrow$ $(\prime a,\prime b,\prime c)$ *transition list* $\Rightarrow$ $(\prime a,\prime b,\prime c)$ *transition list*$) \Rightarrow$
$((\prime a,\prime b,\prime c)$ *fsm* $\Rightarrow$ $(\prime a,\prime b,\prime c)$ *state-cover-assignment* $\Rightarrow (\prime b\times\prime c)$ *prefix-tree* $\Rightarrow \prime d \Rightarrow (\prime d \Rightarrow (\prime b\times\prime c)$ *list* $\Rightarrow \prime d) \Rightarrow (\prime d \Rightarrow (\prime b\times\prime c)$ *list* $\Rightarrow (\prime b\times\prime c)$ *list list*$) \Rightarrow (\prime d \Rightarrow (\prime b\times\prime c)$ *list* $\Rightarrow (\prime b\times\prime c)$ *list* $\Rightarrow \prime d) \Rightarrow$ *nat* $\Rightarrow (\prime a,\prime b,\prime c)$ *transition* $\Rightarrow$ $(\prime a,\prime b,\prime c)$ *transition list* $\Rightarrow$ ($(\prime a,\prime b,\prime c)$ *transition list* $\times$ $(\prime b\times\prime c)$ *prefix-tree* $\times \prime d)) \Rightarrow$

$((\prime a,\prime b,\prime c)$ *fsm* $\Rightarrow$ $(\prime a,\prime b,\prime c)$ *state-cover-assignment* $\Rightarrow (\prime b\times\prime c)$ *prefix-tree* $\Rightarrow \prime d \Rightarrow (\prime d \Rightarrow (\prime b\times\prime c)$ *list* $\Rightarrow \prime d) \Rightarrow (\prime d \Rightarrow (\prime b\times\prime c)$ *list* $\Rightarrow (\prime b\times\prime c)$ *list list*$) \Rightarrow \prime a \Rightarrow \prime b \Rightarrow \prime c \Rightarrow ((\prime b\times\prime c)$ *prefix-tree*$) \times \prime d) \Rightarrow$
$((\prime a,\prime b,\prime c)$ *fsm* $\Rightarrow (\prime b\times\prime c)$ *prefix-tree* $\Rightarrow \prime d) \Rightarrow$
$(\prime d \Rightarrow (\prime b\times\prime c)$ *list* $\Rightarrow \prime d) \Rightarrow$
$(\prime d \Rightarrow (\prime b\times\prime c)$ *list* $\Rightarrow (\prime b\times\prime c)$ *list list*$) \Rightarrow$
$(\prime d \Rightarrow (\prime b\times\prime c)$ *list* $\Rightarrow (\prime b\times\prime c)$ *list* $\Rightarrow \prime d) \Rightarrow$
nat $\Rightarrow$
$(\prime b\times\prime c)$ *prefix-tree*

where
h-framework M
get-state-cover
handle-state-cover
sort-transitions
handle-unverified-transition
handle-unverified-io-pair
cg-initial
cg-insert
cg-lookup
cg-merge
m
= (*let*
rstates-set = *reachable-states M*;
rstates = *reachable-states-as-list M*;
rstates-io = *List.product rstates* (*List.product* (*inputs-as-list M*) (*outputs-as-list M*));
undefined-io-pairs = *List.filter* (λ (*q*,(*x*,*y*)) . *h-obs M q x y* = *None*) *rstates-io*;
V = *get-state-cover M*;
TG1 = *handle-state-cover M V cg-initial cg-insert cg-lookup*;
sc-covered-transitions = (⋃ *q* ∈ *rstates-set* . *covered-transitions M V* (*V q*));
unverified-transitions = *sort-transitions M V* (*filter* (λ*t* . *t-source t* ∈ *rstates-set* ∧ *t* ∉ *sc-covered-transitions*) (*transitions-as-list M*));
verify-transition = (λ (*X*,*T*,*G*) *t* . *handle-unverified-transition M V T G cg-insert cg-lookup cg-merge m t X*);
TG2 = *snd* (*foldl verify-transition* (*unverified-transitions*, *TG1*) *unverified-transitions*);
verify-undefined-io-pair = (λ *T* (*q*,(*x*,*y*)) . *fst* (*handle-unverified-io-pair M V*

T (*snd TG2*) *cg-insert cg-lookup q x y*))
in
foldl verify-undefined-io-pair (*fst TG2*) *undefined-io-pairs*)

18.3 Required Conditions on Procedural Parameters

definition *separates-state-cover* :: (('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ ('*a*,'*b*,'*c*) *state-cover-assignment* ⇒ (('*a*,'*b*,'*c*) *fsm* ⇒ ('*b*×'*c*) *prefix-tree* ⇒ '*d*) ⇒ ('*d* ⇒ ('*b*×'*c*) *list* ⇒ '*d*) ⇒ ('*d* ⇒ ('*b*×'*c*) *list* ⇒ ('*b*×'*c*) *list list*) ⇒ ((('*b*×'*c*) *prefix-tree* × '*d*)) ⇒
('*a*,'*b*,'*c*) *fsm* ⇒
('*e*,'*b*,'*c*) *fsm* ⇒
(('*a*,'*b*,'*c*) *fsm* ⇒ ('*b*×'*c*) *prefix-tree* ⇒ '*d*) ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ '*d*) ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ ('*b*×'*c*) *list list*) ⇒
bool
where
separates-state-cover f M1 M2 cg-initial cg-insert cg-lookup =
(∀ *V* .
(*V* ' *reachable-states M1* ⊆ *set* (*fst* (*f M1 V cg-initial cg-insert cg-lookup*)))
∧ *finite-tree* (*fst* (*f M1 V cg-initial cg-insert cg-lookup*))
∧ (*observable M1* ⟶
observable M2 ⟶
minimal M1 ⟶
minimal M2 ⟶
inputs M2 = *inputs M1* ⟶
outputs M2 = *outputs M1* ⟶
is-state-cover-assignment M1 V ⟶
convergence-graph-insert-invar M1 M2 cg-lookup cg-insert ⟶
convergence-graph-initial-invar M1 M2 cg-lookup cg-initial ⟶
L M1 ∩ *set* (*fst* (*f M1 V cg-initial cg-insert cg-lookup*)) = *L M2* ∩ *set* (*fst* (*f M1 V cg-initial cg-insert cg-lookup*)) ⟶
(*preserves-divergence M1 M2* (*V* ' *reachable-states M1*)
∧ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*f M1 V cg-initial cg-insert cg-lookup*)))))

definition *handles-transition* :: (('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒
('*a*,'*b*,'*c*) *state-cover-assignment* ⇒
('*b*×'*c*) *prefix-tree* ⇒
'*d* ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ '*d*) ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ ('*b*×'*c*) *list list*) ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ ('*b*×'*c*) *list* ⇒ '*d*) ⇒
nat ⇒
('*a*,'*b*,'*c*) *transition* ⇒
('*a*,'*b*,'*c*) *transition list* ⇒
(('*a*,'*b*,'*c*) *transition list* × ('*b*×'*c*) *prefix-tree* × '*d*))
⇒

$('a::linorder,'b::linorder,'c::linorder)$ *fsm* ⇒
$('e,'b,'c)$ *fsm* ⇒
$('a,'b,'c)$ *state-cover-assignment* ⇒
$('b\times'c)$ *prefix-tree* ⇒
$('d \Rightarrow ('b\times'c)$ *list* $\Rightarrow 'd) \Rightarrow$
$('d \Rightarrow ('b\times'c)$ *list* $\Rightarrow ('b\times'c)$ *list list*$) \Rightarrow$
$('d \Rightarrow ('b\times'c)$ *list* $\Rightarrow ('b\times'c)$ *list* $\Rightarrow 'd) \Rightarrow$
bool

where

handles-transition f M1 M2 V T0 cg-insert cg-lookup cg-merge =
(∀ *T G m t X* .
(*set T* ⊆ *set* (*fst* (*snd* (*f M1 V T G cg-insert cg-lookup cg-merge m t X*))))
∧ (*finite-tree T* ⟶ *finite-tree* (*fst* (*snd* (*f M1 V T G cg-insert cg-lookup cg-merge m t X*))))
∧ (*observable M1* ⟶
observable M2 ⟶
minimal M1 ⟶
minimal M2 ⟶
size-r M1 ≤ *m* ⟶
size M2 ≤ *m* ⟶
inputs M2 = *inputs M1* ⟶
outputs M2 = *outputs M1* ⟶
is-state-cover-assignment M1 V ⟶
preserves-divergence M1 M2 (*V* ‘ *reachable-states M1*) ⟶
V ‘ *reachable-states M1* ⊆ *set T* ⟶
t ∈ *transitions M1* ⟶
t-source t ∈ *reachable-states M1* ⟶
((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]) ≠ (*V* (*t-target t*)) ⟶
convergence-graph-lookup-invar M1 M2 cg-lookup G ⟶
convergence-graph-insert-invar M1 M2 cg-lookup cg-insert ⟶
convergence-graph-merge-invar M1 M2 cg-lookup cg-merge ⟶
L M1 ∩ *set* (*fst* (*snd* (*f M1 V T G cg-insert cg-lookup cg-merge m t X*))) = *L M2* ∩ *set* (*fst* (*snd* (*f M1 V T G cg-insert cg-lookup cg-merge m t X*))) ⟶
(*set T0* ⊆ *set T*) ⟶
(∀ γ . (*length* γ ≤ (*m*−*size-r M1*) ∧ *list.set* γ ⊆ *inputs M1* × *outputs M1* ∧ *butlast* γ ∈ *LS M1* (*t-target t*))
⟶ ((*L M1* ∩ (*V* ‘ *reachable-states M1* ∪ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω′ | ω′. ω′ ∈ *list.set* (*prefixes* γ)})
= *L M2* ∩ (*V* ‘ *reachable-states M1* ∪ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω′ | ω′. ω′ ∈ *list.set* (*prefixes* γ)}))
∧ *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1* ∪ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω′ | ω′. ω′ ∈ *list.set* (*prefixes* γ)})))
∧ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*snd* (*f M1 V T G cg-insert cg-lookup cg-merge m t X*)))))

definition *handles-io-pair* :: $(('a::linorder,'b::linorder,'c::linorder)$ *fsm* ⇒
$('a,'b,'c)$ *state-cover-assignment* ⇒
$('b\times'c)$ *prefix-tree* ⇒

$'d \Rightarrow$
$('d \Rightarrow ('b\times'c)$ *list* $\Rightarrow 'd) \Rightarrow$
$('d \Rightarrow ('b\times'c)$ *list* $\Rightarrow ('b\times'c)$ *list list*$) \Rightarrow$
$'a \Rightarrow 'b \Rightarrow 'c \Rightarrow$
$(('b\times'c)$ *prefix-tree* $\times$ $'d)) \Rightarrow$
$('a$::*linorder*,$'b$::*linorder*,$'c$::*linorder*) *fsm* $\Rightarrow$
$('e,'b,'c)$ *fsm* $\Rightarrow$
$('d \Rightarrow ('b\times'c)$ *list* $\Rightarrow 'd) \Rightarrow$
$('d \Rightarrow ('b\times'c)$ *list* $\Rightarrow ('b\times'c)$ *list list*$) \Rightarrow$

bool

where
handles-io-pair f M1 M2 cg-insert cg-lookup =
($\forall$ *V T G q x y* .
(*set T* $\subseteq$ *set* (*fst* (*f M1 V T G cg-insert cg-lookup q x y*)))
$\wedge$ (*finite-tree T* $\longrightarrow$ *finite-tree* (*fst* (*f M1 V T G cg-insert cg-lookup q x y*)))
$\wedge$ (*observable M1* $\longrightarrow$
observable M2 $\longrightarrow$
minimal M1 $\longrightarrow$
minimal M2 $\longrightarrow$
inputs M2 = *inputs M1* $\longrightarrow$
outputs M2 = *outputs M1* $\longrightarrow$
is-state-cover-assignment M1 V $\longrightarrow$
L M1 $\cap$ (*V* ‘ *reachable-states M1*) = *L M2* $\cap$ *V* ‘ *reachable-states M1* $\longrightarrow$
q $\in$ *reachable-states M1* $\longrightarrow$
x $\in$ *inputs M1* $\longrightarrow$
y $\in$ *outputs M1* $\longrightarrow$
convergence-graph-lookup-invar M1 M2 cg-lookup G $\longrightarrow$
convergence-graph-insert-invar M1 M2 cg-lookup cg-insert $\longrightarrow$
L M1 $\cap$ *set* (*fst* (*f M1 V T G cg-insert cg-lookup q x y*)) = *L M2* $\cap$ *set* (*fst* (*f M1 V T G cg-insert cg-lookup q x y*)) $\longrightarrow$
(*L M1* $\cap$ {(*V q*)@[(*x*,*y*)]} = *L M2* $\cap$ {(*V q*)@[(*x*,*y*)]})
$\wedge$ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*f M1 V T G cg-insert cg-lookup q x y*))))

18.4 Completeness and Finiteness of the Scheme

lemma *unverified-transitions-handle-all-transitions* :
assumes *observable M1*
and *is-state-cover-assignment M1 V*
and *L M1* $\cap$ *V* ‘ *reachable-states M1* = *L M2* $\cap$ *V* ‘ *reachable-states M1*
and *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)
and *handles-unverified-transitions*: $\bigwedge$ *t* γ . *t* $\in$ *transitions M1* $\Longrightarrow$
t-source t $\in$ *reachable-states M1* $\Longrightarrow$
length $\gamma \leq k \Longrightarrow$
list.set $\gamma \subseteq$ *inputs M1* $\times$ *outputs M1* $\Longrightarrow$
butlast $\gamma \in$ *LS M1* (*t-target t*) $\Longrightarrow$
(*V* (*t-target t*) $\neq$ (*V* (*t-source t*))@[(*t-input t*,

t-output t)]) $\Longrightarrow$
((*L M1* ∩ (*V* ‘ *reachable-states M1* ∪ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω′ | ω′. ω′ ∈ *list.set* (*prefixes* γ)})
= *L M2* ∩ (*V* ‘ *reachable-states M1* ∪ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω′ | ω′. ω′ ∈ *list.set* (*prefixes* γ)}))
∧ *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1* ∪ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω′ | ω′. ω′ ∈ *list.set* (*prefixes* γ)}))
and *handles-undefined-io-pairs*: ⋀ *q x y* . *q* ∈ *reachable-states M1* $\Longrightarrow$ *x* ∈ *inputs M1* $\Longrightarrow$ *y* ∈ *outputs M1* $\Longrightarrow$ *h-obs M1 q x y* = *None* $\Longrightarrow$ *L M1* ∩ {*V q* @ [(*x*,*y*)]} = *L M2* ∩ {*V q* @ [(*x*,*y*)]}
and *t* ∈ *transitions M1*
and *t-source t* ∈ *reachable-states M1*
and *length* γ ≤ *k*
and *list.set* γ ⊆ *inputs M1* × *outputs M1*
and *butlast* γ ∈ *LS M1* (*t-target t*)
shows (*L M1* ∩ (*V* ‘ *reachable-states M1* ∪ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω′ | ω′. ω′ ∈ *list.set* (*prefixes* γ)})
= *L M2* ∩ (*V* ‘ *reachable-states M1* ∪ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω′ | ω′. ω′ ∈ *list.set* (*prefixes* γ)}))
∧ *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1* ∪ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω′ | ω′. ω′ ∈ *list.set* (*prefixes* γ)})
⟨*proof*⟩

lemma *abstract-h-condition-by-transition-and-io-pair-coverage* :
assumes *observable M1*
and *is-state-cover-assignment M1 V*
and *L M1* ∩ *V* ‘ *reachable-states M1* = *L M2* ∩ *V* ‘ *reachable-states M1*
and *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)
and *handles-unverified-transitions*: ⋀ *t* γ . *t* ∈ *transitions M1* $\Longrightarrow$
t-source t ∈ *reachable-states M1* $\Longrightarrow$
length γ ≤ *k* $\Longrightarrow$
list.set γ ⊆ *inputs M1* × *outputs M1* $\Longrightarrow$
butlast γ ∈ *LS M1* (*t-target t*) $\Longrightarrow$
((*L M1* ∩ (*V* ‘ *reachable-states M1* ∪ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω′ | ω′. ω′ ∈ *list.set* (*prefixes* γ)})
= *L M2* ∩ (*V* ‘ *reachable-states M1* ∪ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω′ | ω′. ω′ ∈ *list.set* (*prefixes* γ)}))
∧ *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1* ∪ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω′ | ω′. ω′ ∈ *list.set* (*prefixes* γ)}))
and *handles-undefined-io-pairs*: ⋀ *q x y* . *q* ∈ *reachable-states M1* $\Longrightarrow$ *x* ∈ *inputs M1* $\Longrightarrow$ *y* ∈ *outputs M1* $\Longrightarrow$ *h-obs M1 q x y* = *None* $\Longrightarrow$ *L M1* ∩ {*V q* @ [(*x*,*y*)]} = *L M2* ∩ {*V q* @ [(*x*,*y*)]}
and *q* ∈ *reachable-states M1*
and *length* γ ≤ *Suc k*
and *list.set* γ ⊆ *inputs M1* × *outputs M1*
and *butlast* γ ∈ *LS M1 q*
shows (*L M1* ∩ (*V* ‘ *reachable-states M1* ∪ {*V q* @ ω′ | ω′. ω′ ∈ *list.set* (*prefixes*

γ)})
= *L M2* ∩ (*V* ‘ *reachable-states M1* ∪ {*V q* @ ω′ | ω′. ω′ ∈ *list.set* (*prefixes* γ)}))
∧ *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1* ∪ {*V q* @ ω′ | ω′. ω′ ∈ *list.set* (*prefixes* γ)})
⟨*proof*⟩

lemma *abstract-h-condition-by-unverified-transition-and-io-pair-coverage* :
assumes *observable M1*
and *is-state-cover-assignment M1 V*
and *L M1* ∩ *V* ‘ *reachable-states M1* = *L M2* ∩ *V* ‘ *reachable-states M1*
and *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)
and *handles-unverified-transitions*: ⋀ *t* γ . *t* ∈ *transitions M1* ⟹
t-source t ∈ *reachable-states M1* ⟹
length γ ≤ *k* ⟹
list.set γ ⊆ *inputs M1* × *outputs M1* ⟹
butlast γ ∈ *LS M1* (*t-target t*) ⟹
(*V* (*t-target t*) ≠ (*V* (*t-source t*))@[(*t-input t*, *t-output t*)]) ⟹
((*L M1* ∩ (*V* ‘ *reachable-states M1* ∪ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω′ | ω′. ω′ ∈ *list.set* (*prefixes* γ)})
= *L M2* ∩ (*V* ‘ *reachable-states M1* ∪ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω′ | ω′. ω′ ∈ *list.set* (*prefixes* γ)}))
∧ *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1* ∪ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω′ | ω′. ω′ ∈ *list.set* (*prefixes* γ)}))
and *handles-undefined-io-pairs*: ⋀ *q x y* . *q* ∈ *reachable-states M1* ⟹ *x* ∈ *inputs M1* ⟹ *y* ∈ *outputs M1* ⟹ *h-obs M1 q x y* = *None* ⟹ *L M1* ∩ {*V q* @ [(*x*,*y*)]} = *L M2* ∩ {*V q* @ [(*x*,*y*)]}
and *q* ∈ *reachable-states M1*
and *length* γ ≤ *Suc k*
and *list.set* γ ⊆ *inputs M1* × *outputs M1*
and *butlast* γ ∈ *LS M1 q*
shows (*L M1* ∩ (*V* ‘ *reachable-states M1* ∪ {*V q* @ ω′ | ω′. ω′ ∈ *list.set* (*prefixes* γ)})
= *L M2* ∩ (*V* ‘ *reachable-states M1* ∪ {*V q* @ ω′ | ω′. ω′ ∈ *list.set* (*prefixes* γ)}))
∧ *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1* ∪ {*V q* @ ω′ | ω′. ω′ ∈ *list.set* (*prefixes* γ)})
⟨*proof*⟩

lemma *h-framework-completeness-and-finiteness* :
fixes *M1* :: (′*a*::*linorder*,′*b*::*linorder*,′*c*::*linorder*) *fsm*
fixes *M2* :: (′*e*,′*b*,′*c*) *fsm*
fixes *cg-insert* :: (′*d* ⇒ (′*b*×′*c*) *list* ⇒ ′*d*)
assumes *observable M1*
and *observable M2*

and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
and *is-state-cover-assignment M1* (*get-state-cover M1*)
and ⋀ *xs* . *List.set xs* = *List.set* (*sort-transitions M1* (*get-state-cover M1*) *xs*)
and *convergence-graph-initial-invar M1 M2 cg-lookup cg-initial*
and *convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*
and *convergence-graph-merge-invar M1 M2 cg-lookup cg-merge*
and *separates-state-cover handle-state-cover M1 M2 cg-initial cg-insert cg-lookup*
and *handles-transition handle-unverified-transition M1 M2* (*get-state-cover M1*) (*fst* (*handle-state-cover M1* (*get-state-cover M1*) *cg-initial cg-insert cg-lookup*)) *cg-insert cg-lookup cg-merge*
and *handles-io-pair handle-unverified-io-pair M1 M2 cg-insert cg-lookup*
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*h-framework M1 get-state-cover handle-state-cover sort-transitions handle-unverified-transition handle-unverified-io-pair cg-initial cg-insert cg-lookup cg-merge m*))
= (*L M2* ∩ *set* (*h-framework M1 get-state-cover handle-state-cover sort-transitions handle-unverified-transition handle-unverified-io-pair cg-initial cg-insert cg-lookup cg-merge m*)))
(**is** (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*)))
and *finite-tree* (*h-framework M1 get-state-cover handle-state-cover sort-transitions handle-unverified-transition handle-unverified-io-pair cg-initial cg-insert cg-lookup cg-merge m*)
⟨*proof*⟩

end

19 SPY-Framework

This theory defines the SPY-Framework and provides completeness properties.

theory *SPY-Framework*
imports *H-Framework*
begin

19.1 Definition of the Framework

definition *spy-framework* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒
(('*a*,'*b*,'*c*) *fsm* ⇒ ('*a*,'*b*,'*c*) *state-cover-assignment*) ⇒

(('*a*,'*b*,'*c*) *fsm* ⇒ ('*a*,'*b*,'*c*) *state-cover-assignment* ⇒ (('*a*,'*b*,'*c*) *fsm* ⇒ ('*b*×'*c*) *prefix-tree* ⇒ '*d*) ⇒ ('*d* ⇒ ('*b*×'*c*) *list* ⇒ '*d*) ⇒ ('*d* ⇒ ('*b*×'*c*) *list* ⇒ ('*b*×'*c*) *list list*) ⇒ (('*b*×'*c*) *prefix-tree* × '*d*)) ⇒

$(('a,'b,'c)$ *fsm* $\Rightarrow$ $('a,'b,'c)$ *state-cover-assignment* $\Rightarrow$ $('a,'b,'c)$ *transition list* $\Rightarrow$ $('a,'b,'c)$ *transition list*$)$ $\Rightarrow$
$(('a,'b,'c)$ *fsm* $\Rightarrow$ $('a,'b,'c)$ *state-cover-assignment* $\Rightarrow$ $('b \times 'c)$ *prefix-tree* $\Rightarrow$ $'d$ $\Rightarrow$ $('d \Rightarrow ('b \times 'c)$ *list* $\Rightarrow 'd)$ $\Rightarrow$ $('d \Rightarrow ('b \times 'c)$ *list* $\Rightarrow ('b \times 'c)$ *list list*$)$ $\Rightarrow$ *nat* $\Rightarrow$ $('a,'b,'c)$ *transition* $\Rightarrow$ $(('b \times 'c)$ *prefix-tree* $\times$ $'d))$ $\Rightarrow$
$(('a,'b,'c)$ *fsm* $\Rightarrow$ $('a,'b,'c)$ *state-cover-assignment* $\Rightarrow$ $('b \times 'c)$ *prefix-tree* $\Rightarrow$ $'d$ $\Rightarrow$ $('d \Rightarrow ('b \times 'c)$ *list* $\Rightarrow 'd)$ $\Rightarrow$ $('d \Rightarrow ('b \times 'c)$ *list* $\Rightarrow ('b \times 'c)$ *list list*$)$ $\Rightarrow$ $'a \Rightarrow 'b \Rightarrow 'c \Rightarrow$ $(('b \times 'c)$ *prefix-tree*$)$ $\times$ $'d)$ $\Rightarrow$
$(('a,'b,'c)$ *fsm* $\Rightarrow$ $('b \times 'c)$ *prefix-tree* $\Rightarrow$ $'d)$ $\Rightarrow$
$('d \Rightarrow ('b \times 'c)$ *list* $\Rightarrow 'd)$ $\Rightarrow$
$('d \Rightarrow ('b \times 'c)$ *list* $\Rightarrow ('b \times 'c)$ *list list*$)$ $\Rightarrow$
$('d \Rightarrow ('b \times 'c)$ *list* $\Rightarrow ('b \times 'c)$ *list* $\Rightarrow 'd)$ $\Rightarrow$
nat $\Rightarrow$
$('b \times 'c)$ *prefix-tree*

where

spy-framework M
get-state-cover
separate-state-cover
sort-unverified-transitions
establish-convergence
append-io-pair
cg-initial
cg-insert
cg-lookup
cg-merge
m
= (*let*
rstates-set = *reachable-states M*;
rstates = *reachable-states-as-list M*;
rstates-io = *List.product rstates* (*List.product* (*inputs-as-list M*) (*outputs-as-list M*));
undefined-io-pairs = *List.filter* (λ (*q*,(*x*,*y*)) . *h-obs M q x y* = *None*) *rstates-io*;
V = *get-state-cover M*;
n = *size-r M*;
TG1 = *separate-state-cover M V cg-initial cg-insert cg-lookup*;
sc-covered-transitions = (⋃ *q* ∈ *rstates-set* . *covered-transitions M V* (*V q*));
unverified-transitions = *sort-unverified-transitions M V* (*filter* (λ*t* . *t-source t* ∈ *rstates-set* ∧ *t* ∉ *sc-covered-transitions*) (*transitions-as-list M*));
verify-transition = (λ (*T*,*G*) *t* . *let TGxy* = *append-io-pair M V T G cg-insert cg-lookup* (*t-source t*) (*t-input t*) (*t-output t*);
(T',G') = *establish-convergence M V* (*fst TGxy*) (*snd TGxy*) *cg-insert cg-lookup m t*;
G'' = *cg-merge* G' ((*V* (*t-source t*)) @ [(*t-input t*, *t-output t*)]) (*V* (*t-target t*))
in (T',G''));
TG2 = *foldl verify-transition TG1 unverified-transitions*;
verify-undefined-io-pair = (λ *T* (*q*,(*x*,*y*)) . *fst* (*append-io-pair M V T* (*snd TG2*) *cg-insert cg-lookup q x y*))
in

foldl verify-undefined-io-pair (fst TG2) undefined-io-pairs)

19.2 Required Conditions on Procedural Parameters

definition *verifies-transition* :: ((′a::linorder,′b::linorder,′c::linorder) fsm ⇒
('a,'b,'c) state-cover-assignment ⇒
('b×'c) prefix-tree ⇒
'd ⇒
('d ⇒ ('b×'c) list ⇒ 'd) ⇒
('d ⇒ ('b×'c) list ⇒ ('b×'c) list list) ⇒
nat ⇒
('a,'b,'c) transition ⇒
(('b×'c) prefix-tree × 'd)) ⇒
('a::linorder,'b::linorder,'c::linorder) fsm ⇒
('e,'b,'c) fsm ⇒
('a,'b,'c) state-cover-assignment ⇒
('b×'c) prefix-tree ⇒
('d ⇒ ('b×'c) list ⇒ 'd) ⇒
('d ⇒ ('b×'c) list ⇒ ('b×'c) list list) ⇒
bool

where

verifies-transition f M1 M2 V T0 cg-insert cg-lookup =
(∀ T G m t .
(set T ⊆ set (fst (f M1 V T G cg-insert cg-lookup m t)))
∧ (finite-tree T ⟶ finite-tree (fst (f M1 V T G cg-insert cg-lookup m t)))
∧ (observable M1 ⟶
observable M2 ⟶
minimal M1 ⟶
minimal M2 ⟶
size-r M1 ≤ m ⟶
size M2 ≤ m ⟶
inputs M2 = inputs M1 ⟶
outputs M2 = outputs M1 ⟶
is-state-cover-assignment M1 V ⟶
preserves-divergence M1 M2 (V ‘ reachable-states M1) ⟶
V ‘ reachable-states M1 ⊆ set T ⟶
t ∈ transitions M1 ⟶
t-source t ∈ reachable-states M1 ⟶
((V (t-source t)) @ [(t-input t,t-output t)]) ≠ (V (t-target t)) ⟶
((V (t-source t)) @ [(t-input t,t-output t)]) ∈ L M2 ⟶
convergence-graph-lookup-invar M1 M2 cg-lookup G ⟶
convergence-graph-insert-invar M1 M2 cg-lookup cg-insert ⟶
L M1 ∩ set (fst (f M1 V T G cg-insert cg-lookup m t)) = L M2 ∩ set (fst (f M1 V T G cg-insert cg-lookup m t)) ⟶
(set T0 ⊆ set T) ⟶
(converge M2 ((V (t-source t)) @ [(t-input t,t-output t)]) (V (t-target t)))
∧ convergence-graph-lookup-invar M1 M2 cg-lookup (snd (f M1 V T G cg-insert cg-lookup m t))))

definition *verifies-io-pair* :: (('a::linorder,'b::linorder,'c::linorder) fsm ⇒
('a,'b,'c) state-cover-assignment ⇒
('b×'c) prefix-tree ⇒
'd ⇒
('d ⇒ ('b×'c) list ⇒ 'd) ⇒
('d ⇒ ('b×'c) list ⇒ ('b×'c) list list) ⇒
'a ⇒ 'b ⇒ 'c ⇒
(('b×'c) prefix-tree × 'd)) ⇒
('a::linorder,'b::linorder,'c::linorder) fsm ⇒
('e,'b,'c) fsm ⇒
('d ⇒ ('b×'c) list ⇒ 'd) ⇒
('d ⇒ ('b×'c) list ⇒ ('b×'c) list list) ⇒

bool

where
verifies-io-pair f M1 M2 cg-insert cg-lookup =
(∀ V T G q x y .
(set T ⊆ set (fst (f M1 V T G cg-insert cg-lookup q x y)))
∧ (finite-tree T ⟶ finite-tree (fst (f M1 V T G cg-insert cg-lookup q x y)))
∧ (observable M1 ⟶
observable M2 ⟶
minimal M1 ⟶
minimal M2 ⟶
inputs M2 = inputs M1 ⟶
outputs M2 = outputs M1 ⟶
is-state-cover-assignment M1 V ⟶
L M1 ∩ (V ' reachable-states M1) = L M2 ∩ V ' reachable-states M1
⟶
q ∈ reachable-states M1 ⟶
x ∈ inputs M1 ⟶
y ∈ outputs M1 ⟶
convergence-graph-lookup-invar M1 M2 cg-lookup G ⟶
convergence-graph-insert-invar M1 M2 cg-lookup cg-insert ⟶
L M1 ∩ set (fst (f M1 V T G cg-insert cg-lookup q x y)) = L M2 ∩ set
(fst (f M1 V T G cg-insert cg-lookup q x y)) ⟶
(∃ α .
converge M1 α (V q) ∧
converge M2 α (V q) ∧
α ∈ set (fst (f M1 V T G cg-insert cg-lookup q x y)) ∧
α@[(x,y)] ∈ set (fst (f M1 V T G cg-insert cg-lookup q x y)))
∧ convergence-graph-lookup-invar M1 M2 cg-lookup (snd (f M1 V T G
cg-insert cg-lookup q x y))))

lemma *verifies-io-pair-handled*:
assumes *verifies-io-pair f M1 M2 cg-insert cg-lookup*
shows *handles-io-pair f M1 M2 cg-insert cg-lookup*
⟨*proof*⟩

19.3 Completeness and Finiteness of the Framework

lemma *spy-framework-completeness-and-finiteness* :
fixes *M1* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm*
fixes *M2* :: (*'d*,*'b*,*'c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
and *is-state-cover-assignment M1* (*get-state-cover M1*)
and ⋀ *xs* . *List.set xs* = *List.set* (*sort-unverified-transitions M1* (*get-state-cover M1*) *xs*)
and *convergence-graph-initial-invar M1 M2 cg-lookup cg-initial*
and *convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*
and *convergence-graph-merge-invar M1 M2 cg-lookup cg-merge*
and *separates-state-cover separate-state-cover M1 M2 cg-initial cg-insert cg-lookup*
and *verifies-transition establish-convergence M1 M2* (*get-state-cover M1*) (*fst* (*separate-state-cover M1* (*get-state-cover M1*) *cg-initial cg-insert cg-lookup*)) *cg-insert cg-lookup*
and *verifies-io-pair append-io-pair M1 M2 cg-insert cg-lookup*
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*spy-framework M1 get-state-cover separate-state-cover sort-unverified-transitions establish-convergence append-io-pair cg-initial cg-insert cg-lookup cg-merge m*))
= (*L M2* ∩ *set* (*spy-framework M1 get-state-cover separate-state-cover sort-unverified-transitions establish-convergence append-io-pair cg-initial cg-insert cg-lookup cg-merge m*)))
(**is** (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*)))
and *finite-tree* (*spy-framework M1 get-state-cover separate-state-cover sort-unverified-transitions establish-convergence append-io-pair cg-initial cg-insert cg-lookup cg-merge m*)
⟨*proof*⟩

end

20 Pair-Framework

This theory defines the Pair-Framework and provides completeness properties.

theory *Pair-Framework*
imports *H-Framework*
begin

20.1 Classical H-Condition

definition *satisfies-h-condition* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $('a,'b,'c)$ *state-cover-assignment* $\Rightarrow$ $('b \times 'c)$ *list set* $\Rightarrow$ *nat* $\Rightarrow$ *bool* **where**
satisfies-h-condition M V T m = (*let*
Π = (*V* ‘ *reachable-states M*);
n = *card* (*reachable-states M*);
$\mathcal{X} = \lambda\ q\ .\ \{io@[(x,y)] \mid io\ x\ y\ .\ io \in LS\ M\ q \wedge length\ io \leq m-n \wedge x \in inputs\ M \wedge y \in outputs\ M\}$;
$A = \Pi \times \Pi$;
$B = \Pi \times \{\ (V\ q)\ @\ \tau \mid q\ \tau\ .\ q \in reachable\text{-}states\ M \wedge \tau \in \mathcal{X}\ q\}$;
$C = (\bigcup\ q \in reachable\text{-}states\ M\ .\ \bigcup\ \tau \in \mathcal{X}\ q\ .\ \{\ (V\ q)\ @\ \tau' \mid \tau'\ .\ \tau' \in list.set\ (prefixes\ \tau)\} \times \{(V\ q)@\tau\})$
in
is-state-cover-assignment M V
$\wedge\ \Pi \subseteq T$
$\wedge\ \{\ (V\ q)\ @\ \tau \mid q\ \tau\ .\ q \in reachable\text{-}states\ M \wedge \tau \in \mathcal{X}\ q\} \subseteq T$
$\wedge\ (\forall\ (\alpha,\beta) \in A \cup B \cup C\ .\ \alpha \in L\ M \longrightarrow$
$\beta \in L\ M \longrightarrow$
after-initial M $\alpha \neq$ *after-initial M* $\beta \longrightarrow$
$(\exists\ \omega\ .\ \alpha@\omega \in T\ \wedge$
$\beta@\omega \in T\ \wedge$
distinguishes M (*after-initial M* α) (*after-initial M* β) ω)))

lemma *h-condition-satisfies-abstract-h-condition* :
assumes *observable M*
and *observable I*
and *minimal M*
and *size I* $\leq$ *m*
and *m* $\geq$ *size-r M*
and *inputs I* = *inputs M*
and *outputs I* = *outputs M*
and *satisfies-h-condition M V T m*
and $(L\ M \cap T = L\ I \cap T)$
shows *satisfies-abstract-h-condition M I V m*
⟨*proof*⟩

lemma *h-condition-completeness* :
assumes *observable M*
and *observable I*
and *minimal M*
and *size I* $\leq$ *m*
and *m* $\geq$ *size-r M*
and *inputs I* = *inputs M*
and *outputs I* = *outputs M*
and *satisfies-h-condition M V T m*
shows $(L\ M = L\ I) \longleftrightarrow (L\ M \cap T = L\ I \cap T)$
⟨*proof*⟩

20.2 Helper Functions

fun *language-up-to-length-with-extensions* :: $'a \Rightarrow ('a \Rightarrow 'b \Rightarrow (('c \times 'a)\ list)) \Rightarrow 'b\ list \Rightarrow ('b \times 'c)\ list\ list \Rightarrow nat \Rightarrow ('b \times 'c)\ list\ list$
where
language-up-to-length-with-extensions q hM iM ex 0 = ex |
language-up-to-length-with-extensions q hM iM ex (Suc k) =
ex @ concat (map (λx .concat (map (λ(y,q′) . (map (λp . (x,y) # p)
(language-up-to-length-with-extensions q′ hM iM ex k)))
(hM q x)))
iM)

lemma *language-up-to-length-with-extensions-set* :
assumes $q \in states\ M$
shows *List.set (language-up-to-length-with-extensions q (λ q x . sorted-list-of-set (h M (q,x))) (inputs-as-list M) ex k)*
$= \{io@xy \mid io\ xy\ .\ io \in LS\ M\ q \wedge length\ io \leq k \wedge xy \in List.set\ ex\}$
(**is** *?S1 q k = ?S2 q k*)
⟨*proof*⟩

fun *h-extensions* :: $('a::linorder,'b::linorder,'c::linorder)\ fsm \Rightarrow 'a \Rightarrow nat \Rightarrow ('b \times 'c)\ list\ list$ **where**
h-extensions M q k = (let
iM = inputs-as-list M;
ex = map (λxy . [xy]) (List.product iM (outputs-as-list M));
hM = (λ q x . sorted-list-of-set (h M (q,x)))
in
language-up-to-length-with-extensions q hM iM ex k)

lemma *h-extensions-set* :
assumes $q \in states\ M$
shows *List.set (h-extensions M q k)* $= \{io@[(x,y)] \mid io\ x\ y\ .\ io \in LS\ M\ q \wedge length\ io \leq k \wedge x \in inputs\ M \wedge y \in outputs\ M\}$
⟨*proof*⟩

fun *paths-up-to-length-with-targets* :: $'a \Rightarrow ('a \Rightarrow 'b \Rightarrow (('a,'b,'c)\ transition\ list)) \Rightarrow 'b\ list \Rightarrow nat \Rightarrow (('a,'b,'c)\ path \times 'a)\ list$
where
paths-up-to-length-with-targets q hM iM 0 = [([],q)] |
paths-up-to-length-with-targets q hM iM (Suc k) =
([],q) # (concat (map (λx .concat (map (λt . (map (λ(p,q). (t # p,q))
(paths-up-to-length-with-targets (t-target t) hM iM k)))
(hM q x)))

iM))

lemma *paths-up-to-length-with-targets-set* :
assumes $q \in$ *states M*
shows *List.set* (*paths-up-to-length-with-targets q* (λ *q x* . *map* (λ(y,q') . (q,x,y,q')) (*sorted-list-of-set* (*h M* (q,x)))) (*inputs-as-list M*) k)
= {(p, *target q p*) | p . *path M q p* $\wedge$ *length p* $\leq k$}
(**is** *?S1 q k* = *?S2 q k*)
⟨*proof*⟩

fun *pairs-to-distinguish* :: ($'a$::*linorder*,$'b$::*linorder*,$'c$::*linorder*) *fsm* $\Rightarrow$ ($'a$,$'b$,$'c$) *state-cover-assignment* $\Rightarrow$ ($'a \Rightarrow$ (($'a$,$'b$,$'c$) *path* $\times$ $'a$) *list*) $\Rightarrow$ $'a$ *list* $\Rightarrow$ ((($'b \times$ $'c$) *list* $\times$ $'a$) $\times$ (($'b \times$ $'c$) *list* $\times$ $'a$)) *list* **where**
pairs-to-distinguish M V $\mathcal{X}'$ *rstates* = (*let*
Π = *map* (λq . (V q,q)) *rstates*;
A = *List.product* Π Π;
B = *List.product* Π (*concat* (*map* (λq . *map* (λ (τ,q') . ((V q)@ *p-io* τ,q')) ($\mathcal{X}'$ q)) *rstates*));
C = *concat* (*map* (λq . *concat* (*map* (λ (τ',q'). *map* ($\lambda\tau''$. (((V q)@ *p-io* τ'', *target q* τ''),((V q)@ *p-io* τ',q'))) (*prefixes* τ')) ($\mathcal{X}'$ q))) *rstates*)
in
filter (λ((α,q'),(β,q'')) . $q' \neq q''$) (A@B@C))

lemma *pairs-to-distinguish-elems* :
assumes *observable M*
and *is-state-cover-assignment M V*
and *list.set rstates* = *reachable-states M*
and $\bigwedge$ *q p q'* . $q \in$ *reachable-states M* $\Longrightarrow$ (p,q') $\in$ *list.set* ($\mathcal{X}'$ q) $\longleftrightarrow$ *path M q p* $\wedge$ *target q p* = q' $\wedge$ *length p* $\leq m-n+1$
and ((α,$q1$),(β,$q2$)) $\in$ *list.set* (*pairs-to-distinguish M V* $\mathcal{X}'$ *rstates*)

shows $q1 \in$ *states M* **and** $q2 \in$ *states M* **and** $q1 \neq q2$
and $\alpha \in L\ M$ **and** $\beta \in L\ M$ **and** $q1$ = *after-initial M* α **and** $q2$ = *after-initial M* β
⟨*proof*⟩

lemma *pairs-to-distinguish-containment* :
assumes *observable M*
and *is-state-cover-assignment M V*
and *list.set rstates* = *reachable-states M*
and $\bigwedge$ *q p q'* . $q \in$ *reachable-states M* $\Longrightarrow$ (p,q') $\in$ *list.set* ($\mathcal{X}'$ q) $\longleftrightarrow$ *path M q p* $\wedge$ *target q p* = q' $\wedge$ *length p* $\leq m-n+1$
and (α,β) $\in$ (V ‘ *reachable-states M*) $\times$ (V ‘ *reachable-states M*)
$\cup$ (V ‘ *reachable-states M*) $\times$ { (V q) @ τ | q τ . $q \in$ *reachable-states M* $\wedge$ $\tau \in$ {*io*@[(x,y)] | *io x y* . *io* $\in$ *LS M q* $\wedge$ *length io* $\leq m-n$ $\wedge$ $x \in$ *inputs M* $\wedge$ $y \in$ *outputs M*}}

$\cup$ ($\bigcup$ $q \in$ *reachable-states M* . $\bigcup$ $\tau \in \{io@[(x,y)] \mid io\ x\ y\ .\ io \in LS$ $M\ q \wedge length\ io \leq m{-}n \wedge x \in inputs\ M \wedge y \in outputs\ M\}$. { $(V\ q)$ @ $\tau' \mid \tau'$. $\tau' \in list.set\ (prefixes\ \tau)\} \times \{(V\ q)@\tau\})$
and $\alpha \in L\ M$
and $\beta \in L\ M$
and *after-initial M* $\alpha \neq$ *after-initial M* β
shows $((\alpha,$*after-initial M* $\alpha),(\beta,$*after-initial M* $\beta)) \in$ *list.set* (*pairs-to-distinguish M V* $\mathcal{X}'$ *rstates*)
⟨*proof*⟩

20.3 Definition of the Pair-Framework

definition *pair-framework* :: ('a::linorder,'b::linorder,'c::linorder) *fsm* ⇒
nat ⇒
(('a,'b,'c) *fsm* ⇒ *nat* ⇒ ('b×'c) *prefix-tree*) ⇒
(('a,'b,'c) *fsm* ⇒ *nat* ⇒ ((('b × 'c) *list* × 'a) × (('b × 'c) *list* × 'a)) *list*) ⇒
(('a,'b,'c) *fsm* ⇒ (('b × 'c) *list* × 'a) × ('b × 'c) *list* × 'a ⇒ ('b × 'c) *prefix-tree* ⇒ ('b × 'c) *prefix-tree*) ⇒
('b×'c) *prefix-tree*
where
pair-framework M m get-initial-test-suite get-pairs get-separating-traces =
(*let*
TS = *get-initial-test-suite M m*;
D = *get-pairs M m*;
dist-extension = (λ *t* $((\alpha,q'),(\beta,q''))$. *let tDist* = *get-separating-traces M* $((\alpha,q'),(\beta,q''))$ *t*
in combine-after (*combine-after t* α *tDist*) β *tDist*)
in
foldl dist-extension TS D)

lemma *pair-framework-completeness* :
assumes *observable M*
and *observable I*
and *minimal M*
and *size I* $\leq$ *m*
and *m* $\geq$ *size-r M*
and *inputs I* = *inputs M*
and *outputs I* = *outputs M*
and *is-state-cover-assignment M V*
and $\{(V\ q)@io@[(x,y)] \mid q\ io\ x\ y\ .\ q \in reachable\text{-}states\ M \wedge io \in LS\ M\ q \wedge length\ io \leq m - size\text{-}r\ M \wedge x \in inputs\ M \wedge y \in outputs\ M\} \subseteq set$ (*get-initial-test-suite M m*)
and $\bigwedge \alpha\ \beta\ .\ (\alpha,\beta) \in (V$ ‘ *reachable-states M*) × (*V* ‘ *reachable-states M*)
$\cup$ (*V* ‘ *reachable-states M*) × { $(V\ q)$ @ $\tau \mid q\ \tau\ .\ q \in reachable\text{-}states\ M \wedge \tau \in \{io@[(x,y)] \mid io\ x\ y\ .\ io \in LS\ M\ q \wedge length\ io \leq m{-}size\text{-}r\ M \wedge x \in$

inputs $M \wedge y \in$ *outputs* $M\}\}$
$\cup (\bigcup q \in$ *reachable-states* M . $\bigcup \tau \in \{io@[(x,y)] \mid io\ x\ y\ .\ io \in LS\ M\ q \wedge length\ io \leq m{-}size{-}r\ M \wedge x \in inputs\ M \wedge y \in outputs\ M\}\ .\ \{ (V\ q) @ \tau' \mid \tau'\ .\ \tau' \in list.set\ (prefixes\ \tau)\} \times \{(V\ q)@\tau\}) \Longrightarrow$
$\alpha \in L\ M \Longrightarrow \beta \in L\ M \Longrightarrow$ *after-initial* $M\ \alpha \neq$ *after-initial* $M\ \beta \Longrightarrow$
$((\alpha,$*after-initial* $M\ \alpha),(\beta,$*after-initial* $M\ \beta)) \in$ *list.set* (*get-pairs* M m)
and $\bigwedge \alpha\ \beta\ t\ .\ \alpha \in L\ M \Longrightarrow \beta \in L\ M \Longrightarrow$ *after-initial* $M\ \alpha \neq$ *after-initial* $M\ \beta \Longrightarrow \exists\ io \in$ *set* (*get-separating-traces* M $((\alpha,$*after-initial* $M\ \alpha),(\beta,$*after-initial* $M\ \beta))$ t) $\cup$ (*set* (*after* $t\ \alpha$) $\cap$ *set* (*after* $t\ \beta$)) . *distinguishes* M (*after-initial* $M\ \alpha$) (*after-initial* $M\ \beta$) *io*
shows $(L\ M = L\ I) \longleftrightarrow (L\ M \cap$ *set* (*pair-framework* $M\ m$ *get-initial-test-suite get-pairs get-separating-traces*) $= L\ I \cap$ *set* (*pair-framework* $M\ m$ *get-initial-test-suite get-pairs get-separating-traces*))
⟨*proof*⟩

lemma *pair-framework-finiteness* :
assumes $\bigwedge \alpha\ \beta\ t\ .\ \alpha \in L\ M \Longrightarrow \beta \in L\ M \Longrightarrow$ *after-initial* $M\ \alpha \neq$ *after-initial* $M\ \beta \Longrightarrow$ *finite-tree* (*get-separating-traces* M $((\alpha,$*after-initial* $M\ \alpha),(\beta,$*after-initial* $M\ \beta))$ t)
and *finite-tree* (*get-initial-test-suite* $M\ m$)
and $\bigwedge \alpha\ q'\ \beta\ q''\ .\ ((\alpha,q'),(\beta,q'')) \in$ *list.set* (*get-pairs* $M\ m$) $\Longrightarrow \alpha \in L\ M \wedge \beta \in L\ M \wedge$ *after-initial* $M\ \alpha \neq$ *after-initial* $M\ \beta \wedge q' =$ *after-initial* $M\ \alpha \wedge q'' =$ *after-initial* $M\ \beta$
shows *finite-tree* (*pair-framework* $M\ m$ *get-initial-test-suite get-pairs get-separating-traces*)
⟨*proof*⟩

end

21 Intermediate Implementations

This theory implements various functions to be supplied to the H, SPY, and Pair-Frameworks.

theory *Intermediate-Implementations*
imports *H-Framework SPY-Framework Pair-Framework ../Distinguishability Automatic-Refinement.Misc*
begin

21.1 Functions for the Pair Framework

definition *get-initial-test-suite-H* :: $('a,'b,'c)$ *state-cover-assignment* $\Rightarrow$
$('a{::}linorder,'b{::}linorder,'c{::}linorder)$ *fsm* $\Rightarrow$
nat $\Rightarrow$

('b×'c) *prefix-tree*
where
get-initial-test-suite-H V M m =
(*let*
rstates = *reachable-states-as-list M*;
n = *size-r M*;
iM = *inputs-as-list M*;
T = *from-list* (*concat* (*map* (λ*q* . *map* (λτ. (*V q*)@τ) (*h-extensions M q* (*m*−*n*))) *rstates*))
in T)

lemma *get-initial-test-suite-H-set-and-finite* :
shows {(*V q*)@*io*@[(*x*,*y*)] | *q io x y* . *q* ∈ *reachable-states M* ∧ *io* ∈ *LS M q* ∧ *length io* ≤ *m* − *size-r M* ∧ *x* ∈ *inputs M* ∧ *y* ∈ *outputs M*} ⊆ *set* (*get-initial-test-suite-H V M m*)
and *finite-tree* (*get-initial-test-suite-H V M m*)
⟨*proof*⟩

fun *complete-inputs-to-tree* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ '*a* ⇒ '*c list* ⇒ '*b list* ⇒ ('*b* × '*c*) *prefix-tree* **where**
complete-inputs-to-tree M q ys [] = *Prefix-Tree.empty* |
complete-inputs-to-tree M q ys (*x*#*xs*) = *foldl* (λ *t y* . *case h-obs M q x y of None* ⇒ *insert t* [(*x*,*y*)] |
Some q' ⇒ *combine-after t* [(*x*,*y*)] (*complete-inputs-to-tree M q' ys xs*)) *Prefix-Tree.empty ys*

lemma *complete-inputs-to-tree-finite-tree* :
finite-tree (*complete-inputs-to-tree M q ys xs*)
⟨*proof*⟩

fun *complete-inputs-to-tree-initial* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ '*b list* ⇒ ('*b* × '*c*) *prefix-tree* **where**
complete-inputs-to-tree-initial M xs = *complete-inputs-to-tree M* (*initial M*) (*outputs-as-list M*) *xs*

definition *get-initial-test-suite-H-2* :: *bool* ⇒ ('*a*,'*b*,'*c*) *state-cover-assignment* ⇒
('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒
nat ⇒
('*b*×'*c*) *prefix-tree* **where**
get-initial-test-suite-H-2 c V M m =
(*if c then get-initial-test-suite-H V M m*
else let TS = *get-initial-test-suite-H V M m*;
xss = *map* (*map fst*) (*sorted-list-of-maximal-sequences-in-tree TS*);
ys = *outputs-as-list M*
in

foldl (λ *t xs* . *combine t* (*complete-inputs-to-tree-initial M xs*)) *TS xss*)

lemma *get-initial-test-suite-H-2-set-and-finite* :
shows $\{(V\ q)@io@[(x,y)] \mid q\ io\ x\ y\ .\ q \in \textit{reachable-states}\ M \wedge io \in LS\ M\ q \wedge \textit{length}\ io \leq m - \textit{size-r}\ M \wedge x \in \textit{inputs}\ M \wedge y \in \textit{outputs}\ M\} \subseteq \textit{set}\ (\textit{get-initial-test-suite-H-2}\ c\ V\ M\ m)$ (**is** *?P1*)
and *finite-tree* (*get-initial-test-suite-H-2 c V M m*) (**is** *?P2*)
$\langle proof \rangle$

definition *get-pairs-H* :: ('a,'b,'c) *state-cover-assignment* $\Rightarrow$
('a::*linorder*,'b::*linorder*,'c::*linorder*) *fsm* $\Rightarrow$
nat $\Rightarrow$
((('b $\times$ 'c) *list* $\times$ 'a) $\times$ (('b $\times$ 'c) *list* $\times$ 'a)) *list*
where
get-pairs-H V M m =
(*let*
rstates = *reachable-states-as-list M*;
n = *size-r M*;
iM = *inputs-as-list M*;
hMap = *mapping-of* (*map* (λ(*q*,*x*) . ((*q*,*x*), *map* (λ(*y*,*q'*) . (*q*,*x*,*y*,*q'*)) (*sorted-list-of-set* (*h M* (*q*,*x*))))) (*List.product* (*states-as-list M*) *iM*));
hM = (λ *q x* . *case Mapping.lookup hMap* (*q*,*x*) *of Some ts* $\Rightarrow$ *ts* | *None* $\Rightarrow$ []);
pairs = *pairs-to-distinguish M V* (λ*q* . *paths-up-to-length-with-targets q hM iM* ((*m*−*n*)+1)) *rstates*
in
pairs)

lemma *get-pairs-H-set* :
assumes *observable M*
and *is-state-cover-assignment M V*
shows
$\bigwedge \alpha\ \beta\ .\ (\alpha,\beta) \in (V\ `\ \textit{reachable-states}\ M) \times (V\ `\ \textit{reachable-states}\ M)$
$\cup\ (V\ `\ \textit{reachable-states}\ M) \times \{\ (V\ q)\ @\ \tau \mid q\ \tau\ .\ q \in \textit{reachable-states}\ M \wedge \tau \in \{io@[(x,y)] \mid io\ x\ y\ .\ io \in LS\ M\ q \wedge \textit{length}\ io \leq m{-}\textit{size-r}\ M \wedge x \in \textit{inputs}\ M \wedge y \in \textit{outputs}\ M\}\}$
$\cup\ (\bigcup\ q \in \textit{reachable-states}\ M\ .\ \bigcup\ \tau \in \{io@[(x,y)] \mid io\ x\ y\ .\ io \in LS\ M\ q \wedge \textit{length}\ io \leq m{-}\textit{size-r}\ M \wedge x \in \textit{inputs}\ M \wedge y \in \textit{outputs}\ M\}\ .\ \{\ (V\ q)\ @\ \tau' \mid \tau'\ .\ \tau' \in \textit{list.set}\ (\textit{prefixes}\ \tau)\} \times \{(V\ q)@\tau\}) \Longrightarrow$
$\alpha \in L\ M \Longrightarrow \beta \in L\ M \Longrightarrow \textit{after-initial}\ M\ \alpha \neq \textit{after-initial}\ M\ \beta \Longrightarrow$
$((\alpha,\textit{after-initial}\ M\ \alpha),(\beta,\textit{after-initial}\ M\ \beta)) \in \textit{list.set}\ (\textit{get-pairs-H}\ V\ M\ m)$
and $\bigwedge \alpha\ q'\ \beta\ q''\ .\ ((\alpha,q'),(\beta,q'')) \in \textit{list.set}\ (\textit{get-pairs-H}\ V\ M\ m) \Longrightarrow \alpha \in L\ M \wedge \beta \in L\ M \wedge \textit{after-initial}\ M\ \alpha \neq \textit{after-initial}\ M\ \beta \wedge q' = \textit{after-initial}\ M\ \alpha \wedge q'' = \textit{after-initial}\ M\ \beta$

⟨*proof*⟩

21.2 Functions of the SPYH-Method

21.2.1 Heuristic Functions for Selecting Traces to Extend

fun *estimate-growth* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* ⇒ (*'a* ⇒ *'a* ⇒ (*'b* × *'c*) *list*) ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'c* ⇒ *nat* ⇒ *nat* **where**
  estimate-growth M dist-fun q1 q2 x y errorValue= (*case h-obs M q1 x y of*
    None ⇒ (*case h-obs M q1 x y of*
      None ⇒ *errorValue* |
      Some q2' ⇒ *1*) |
    Some q1' ⇒ (*case h-obs M q2 x y of*
      None ⇒ *1* |
      Some q2' ⇒ *if q1'* = *q2'* ∨ {*q1'*,*q2'*} = {*q1*,*q2*}
        then errorValue
        else 1 + *2* ∗ (*length* (*dist-fun q1 q2*))))

lemma *estimate-growth-result* :
  assumes *observable M*
  and *minimal M*
  and *q1* ∈ *states M*
  and *q2* ∈ *states M*
  and *estimate-growth M dist-fun q1 q2 x y errorValue* < *errorValue*
shows ∃ γ . *distinguishes M q1 q2* ([(*x*,*y*)]@γ)
⟨*proof*⟩

fun *shortest-list-or-default* :: *'a list list* ⇒ *'a list* ⇒ *'a list* **where**
  shortest-list-or-default xs x = *foldl* (λ *a b* . *if length a* < *length b then a else b*) *x xs*

lemma *shortest-list-or-default-elem* :
  shortest-list-or-default xs x ∈ *Set.insert x* (*list.set xs*)
  ⟨*proof*⟩

fun *shortest-list* :: *'a list list* ⇒ *'a list* **where**
  shortest-list [] = *undefined* |
  shortest-list (*x*#*xs*) = *shortest-list-or-default xs x*

lemma *shortest-list-elem* :
  assumes *xs* ≠ []
shows *shortest-list xs* ∈ *list.set xs*
  ⟨*proof*⟩

fun *shortest-list-in-tree-or-default* :: *'a list list* ⇒ *'a prefix-tree* ⇒ *'a list* ⇒ *'a list* **where**
  shortest-list-in-tree-or-default xs T x = *foldl* (λ *a b* . *if isin T a* ∧ *length a* < *length b then a else b*) *x xs*

lemma *shortest-list-in-tree-or-default-elem* :
shortest-list-in-tree-or-default xs T x ∈ *Set.insert x* (*list.set xs*)
⟨*proof*⟩

fun *has-leaf* :: ('b×'c) *prefix-tree* ⇒ 'd ⇒ ('d ⇒ ('b×'c) *list* ⇒ ('b×'c) *list list*) ⇒ ('b×'c) *list* ⇒ *bool* **where**
has-leaf T G cg-lookup α =
(*find* (λ β . *is-maximal-in T* β) (α # *cg-lookup G* α) ≠ *None*)

fun *has-extension* :: ('b×'c) *prefix-tree* ⇒ 'd ⇒ ('d ⇒ ('b×'c) *list* ⇒ ('b×'c) *list list*) ⇒ ('b×'c) *list* ⇒ 'b ⇒ 'c ⇒ *bool* **where**
has-extension T G cg-lookup α *x y* =
(*find* (λ β . *isin T* (β@[(x,y)])) (α # *cg-lookup G* α) ≠ *None*)

fun *get-extension* :: ('b×'c) *prefix-tree* ⇒ 'd ⇒ ('d ⇒ ('b×'c) *list* ⇒ ('b×'c) *list list*) ⇒ ('b×'c) *list* ⇒ 'b ⇒ 'c ⇒ ('b×'c) *list option* **where**
get-extension T G cg-lookup α *x y* =
(*find* (λ β . *isin T* (β@[(x,y)])) (α # *cg-lookup G* α))

fun *get-prefix-of-separating-sequence* :: ('a::*linorder*,'b::*linorder*,'c::*linorder*) *fsm* ⇒ ('b×'c) *prefix-tree* ⇒ 'd ⇒ ('d ⇒ ('b×'c) *list* ⇒ ('b×'c) *list list*) ⇒ ('a ⇒ 'a ⇒ ('b×'c) *list*) ⇒ ('b×'c) *list* ⇒ ('b×'c) *list* ⇒ *nat* ⇒ (*nat* × ('b×'c) *list*) **where**
get-prefix-of-separating-sequence M T G cg-lookup get-distinguishing-trace u v 0 = (*1*,[]) |
get-prefix-of-separating-sequence M T G cg-lookup get-distinguishing-trace u v (*Suc k*)= (*let*
u' = *shortest-list-or-default* (*cg-lookup G u*) *u*;
v' = *shortest-list-or-default* (*cg-lookup G v*) *v*;
su = *after-initial M u*;
sv = *after-initial M v*;
bestPrefix0 = *get-distinguishing-trace su sv*;
minEst0 = *length bestPrefix0* + (*if* (*has-leaf T G cg-lookup u'*) *then 0 else length u'*) + (*if* (*has-leaf T G cg-lookup v'*) *then 0 else length v'*);
errorValue = *Suc minEst0*;
XY = *List.product* (*inputs-as-list M*) (*outputs-as-list M*);
tryIO = (λ (*minEst*,*bestPrefix*) (*x*,*y*) .
if minEst = *0*
then (*minEst*,*bestPrefix*)
else (*case get-extension T G cg-lookup u' x y of*
Some u'' ⇒ (*case get-extension T G cg-lookup v' x y of*
Some v'' ⇒ *if* (*h-obs M su x y* = *None*) ≠ (*h-obs M sv x y* = *None*)

then (0,[])
else if h-obs M su x y = h-obs M sv x y
then (minEst,bestPrefix)
else (let (e,w) = get-prefix-of-separating-sequence M T G cg-lookup get-distinguishing-trace (u''@[(x,y)]) (v''@[(x,y)]) k
in if e = 0
then (0,[])
else if e ≤ minEst
then (e,(x,y)#w)
else (minEst,bestPrefix)) |
None ⇒ (let e = estimate-growth M get-distinguishing-trace su sv x y errorValue;
e' = if e ≠ 1
then if has-leaf T G cg-lookup u''
then e + 1
else if ¬(has-leaf T G cg-lookup (u''@[(x,y)]))
then e + length u' + 1
else e
else e;
e'' = e' + (if ¬(has-leaf T G cg-lookup v') then length v' else 0)
in if e'' ≤ minEst
then (e'',[(x,y)])
else (minEst,bestPrefix))) |
None ⇒ (case get-extension T G cg-lookup v' x y of
Some v'' ⇒ (let e = estimate-growth M get-distinguishing-trace su sv x y errorValue;
e' = if e ≠ 1
then if has-leaf T G cg-lookup v''
then e + 1
else if ¬(has-leaf T G cg-lookup (v''@[(x,y)]))
then e + length v' + 1
else e
else e;
e'' = e' + (if ¬(has-leaf T G cg-lookup u') then length u' else 0)
in if e'' ≤ minEst
then (e'',[(x,y)])
else (minEst,bestPrefix)) |
None ⇒ (minEst,bestPrefix))))
in if ¬ isin T u' ∨ ¬ isin T v'
then (errorValue,[])
else foldl tryIO (minEst0,[]) XY)

lemma *estimate-growth-Suc* :
assumes *errorValue > 0*
shows *estimate-growth M get-distinguishing-trace q1 q2 x y errorValue > 0*
⟨*proof*⟩

lemma *get-extension-result*:
assumes $u \in L\ M1$ **and** $u \in L\ M2$
and *convergence-graph-lookup-invar M1 M2 cg-lookup G*
and *get-extension T G cg-lookup u x y = Some u'*
shows *converge M1 u u'* **and** $u' \in L\ M2 \Longrightarrow$ *converge M2 u u'* **and** $u'@[(x,y)] \in$ *set T*
⟨*proof*⟩

lemma *get-prefix-of-separating-sequence-result* :
fixes *M1* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and $u \in L\ M1$ **and** $u \in L\ M2$
and $v \in L\ M1$ **and** $v \in L\ M2$
and *after-initial M1 u* $\neq$ *after-initial M1 v*
and $\bigwedge \alpha\ \beta\ q1\ q2\ .\ q1 \in states\ M1 \Longrightarrow q2 \in states\ M1 \Longrightarrow q1 \neq q2 \Longrightarrow$ *distinguishes M1 q1 q2* (*get-distinguishing-trace q1 q2*)
and *convergence-graph-lookup-invar M1 M2 cg-lookup G*
and $L\ M1 \cap set\ T = L\ M2 \cap set\ T$
shows *fst* (*get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace u v k*) $= 0 \Longrightarrow \neg$ *converge M2 u v*
and *fst* (*get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace u v k*) $\neq 0 \Longrightarrow \exists\ \gamma$. *distinguishes M1* (*after-initial M1 u*) (*after-initial M1 v*) ((*snd* (*get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace u v k*))@γ)
⟨*proof*⟩

21.2.2 Distributing Convergent Traces

fun *append-heuristic-io* :: (*'b*×*'c*) *prefix-tree* ⇒ (*'b*×*'c*) *list* ⇒ ((*'b*×*'c*) *list* × *int*) ⇒ (*'b*×*'c*) *list* ⇒ ((*'b*×*'c*) *list* × *int*) **where**
append-heuristic-io T w (*uBest*,*lBest*) *u'* = (*let t'* = *after T u'*;
w' = *maximum-prefix t' w*
in if w' = *w*
then (*u'*,*0*::*int*)
else if (*is-maximal-in t' w'* ∧ (*int* (*length w'*) > *lBest* ∨ (*int* (*length w'*) = *lBest* ∧ *length u'* < *length uBest*)))
then (*u'*, *int* (*length w'*))
else (*uBest*,*lBest*))

lemma *append-heuristic-io-in* :
fst (*append-heuristic-io T w* (*uBest*,*lBest*) *u'*) ∈ {*u'*,*uBest*}
⟨*proof*⟩

fun *append-heuristic-input* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ ('*b*×'*c*) *prefix-tree* ⇒ ('*b*×'*c*) *list* ⇒ (('*b*×'*c*) *list* × *int*) ⇒ ('*b*×'*c*) *list* ⇒ (('*b*×'*c*) *list* × *int*) **where**
append-heuristic-input M T w (*uBest*,*lBest*) *u'* = (*let t'* = *after T u'*;
ws = *maximum-fst-prefixes t'* (*map fst w*) (*outputs-as-list M*)
in
foldr (λ *w'* (*uBest'*,*lBest'*::*int*) .
if w' = *w*
then (*u'*,*0*::*int*)
else if (*int* (*length w'*) > *lBest'* ∨ (*int* (*length w'*) = *lBest'* ∧ *length u'* < *length uBest'*))
then (*u'*,*int* (*length w'*))
else (*uBest'*,*lBest'*))
ws (*uBest*,*lBest*))

lemma *append-heuristic-input-in* :
fst (*append-heuristic-input M T w* (*uBest*,*lBest*) *u'*) ∈ {*u'*,*uBest*}
⟨*proof*⟩

fun *distribute-extension* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ ('*b*×'*c*) *prefix-tree* ⇒ '*d* ⇒ ('*d* ⇒ ('*b*×'*c*) *list* ⇒ ('*b*×'*c*) *list list*) ⇒ ('*d* ⇒ ('*b*×'*c*) *list* ⇒ '*d*) ⇒('*b*×'*c*) *list* ⇒ ('*b*×'*c*) *list* ⇒ *bool* ⇒ (('*b*×'*c*) *prefix-tree* ⇒ ('*b*×'*c*) *list* ⇒ (('*b*×'*c*) *list* × *int*) ⇒ ('*b*×'*c*) *list* ⇒ (('*b*×'*c*) *list* × *int*)) ⇒ (('*b*×'*c*) *prefix-tree* ×'*d*) **where**
distribute-extension M T G cg-lookup cg-insert u w completeInputTraces append-heuristic = (*let*
cu = *cg-lookup G u*;
u0 = *shortest-list-in-tree-or-default cu T u*;
l0 = −*1*::*int*;
u' = *fst* ((*foldl* (*append-heuristic T w*) (*u0*,*l0*) (*filter* (*isin T*) *cu*)) :: (('*b*×'*c*) *list* × *int*));
T' = *insert T* (*u'*@*w*);
G' = *cg-insert G* (*maximal-prefix-in-language M* (*initial M*) (*u'*@*w*))
in if completeInputTraces
then let TC = *complete-inputs-to-tree M* (*initial M*) (*outputs-as-list M*) (*map fst* (*u'*@*w*));
T'' = *Prefix-Tree.combine T' TC*
in (*T''*,*G'*)
else (*T'*,*G'*))

lemma *distribute-extension-subset* :
set T ⊆ *set* (*fst* (*distribute-extension M T G cg-lookup cg-insert u w b heuristic*))
⟨*proof*⟩

lemma *distribute-extension-finite* :
assumes *finite-tree T*
shows *finite-tree* (*fst* (*distribute-extension M T G cg-lookup cg-insert u w b heuristic*))
⟨*proof*⟩

lemma *distribute-extension-adds-sequence* :
fixes *M1* :: (′*a*::*linorder*,′*b*::*linorder*,′*c*::*linorder*) *fsm*
assumes *observable M1*
and *minimal M1*
and $u \in L\ M1$ **and** $u \in L\ M2$
and *convergence-graph-lookup-invar M1 M2 cg-lookup G*
and *convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*
and (*L M1* ∩ *set* (*fst* (*distribute-extension M1 T G cg-lookup cg-insert u w b heuristic*)) = *L M2* ∩ *set* (*fst* (*distribute-extension M1 T G cg-lookup cg-insert u w b heuristic*)))
and ⋀ *u*′ *uBest lBest* . *fst* (*heuristic T w* (*uBest*,*lBest*) *u*′) ∈ {*u*′,*uBest*}
shows ∃ *u*′ . *converge M1 u u*′ ∧ *u*′@*w* ∈ *set* (*fst* (*distribute-extension M1 T G cg-lookup cg-insert u w b heuristic*)) ∧ *converge M2 u u*′
and *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*distribute-extension M1 T G cg-lookup cg-insert u w b heuristic*))
⟨*proof*⟩

21.2.3 Distinguishing a Trace from Other Traces

fun *spyh-distinguish* :: (′*a*::*linorder*,′*b*::*linorder*,′*c*::*linorder*) *fsm* ⇒ (′*b*×′*c*) *prefix-tree* ⇒ ′*d* ⇒ (′*d* ⇒ (′*b*×′*c*) *list* ⇒ (′*b*×′*c*) *list list*) ⇒ (′*d* ⇒ (′*b*×′*c*) *list* ⇒ ′*d*) ⇒ (′*a* ⇒ ′*a* ⇒ (′*b*×′*c*) *list*) ⇒ (′*b*×′*c*) *list* ⇒ (′*b*×′*c*) *list list* ⇒ *nat* ⇒ *bool* ⇒ ((′*b*×′*c*) *prefix-tree* ⇒ (′*b*×′*c*) *list* ⇒ ((′*b*×′*c*) *list* × *int*) ⇒ (′*b*×′*c*) *list* ⇒ ((′*b*×′*c*) *list* × *int*)) ⇒ ((′*b*×′*c*) *prefix-tree* × ′*d*) **where**
spyh-distinguish M T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic = (*let*
dist-helper = (λ (*T*,*G*) *v* . *if after-initial M u* = *after-initial M v*
then (*T*,*G*)
else (*let ew* = *get-prefix-of-separating-sequence M T G cg-lookup get-distinguishing-trace u v k*
in if fst ew = *0*
then (*T*,*G*)
else (*let u*′ = (*u*@(*snd ew*));
v′ = (*v*@(*snd ew*));
w′ = *if does-distinguish M* (*after-initial M u*) (*after-initial M v*) (*snd ew*) *then* (*snd ew*) *else* (*snd ew*)@(*get-distinguishing-trace* (*after-initial M u*′) (*after-initial M v*′));
TG′ = *distribute-extension M T G cg-lookup cg-insert u w*′ *completeInputTraces append-heuristic*
in distribute-extension M (*fst TG*′) (*snd TG*′) *cg-lookup cg-insert v w*′ *completeInputTraces append-heuristic*)))

in foldl dist-helper (T,G) X)

lemma *spyh-distinguish-subset* :
set T $\subseteq$ *set* (*fst* (*spyh-distinguish M T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*))
⟨*proof*⟩

lemma *spyh-distinguish-finite* :
fixes *T* :: (*'b*::*linorder*×*'c*::*linorder*) *prefix-tree*
assumes *finite-tree T*
shows *finite-tree* (*fst* (*spyh-distinguish M T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*))
⟨*proof*⟩

lemma *spyh-distinguish-establishes-divergence* :
fixes *M1* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and $u \in L$ *M1* **and** $u \in L$ *M2*
and $\bigwedge$ α β *q1 q2* . *q1* $\in$ *states M1* $\Longrightarrow$ *q2* $\in$ *states M1* $\Longrightarrow$ *q1* $\neq$ *q2* $\Longrightarrow$ *distinguishes M1 q1 q2* (*get-distinguishing-trace q1 q2*)
and *convergence-graph-lookup-invar M1 M2 cg-lookup G*
and *convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*
and *list.set X* $\subseteq$ *L M1*
and *list.set X* $\subseteq$ *L M2*
and *L M1* $\cap$ *set* (*fst* (*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*)) = *L M2* $\cap$ *set* (*fst* (*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*))
and $\bigwedge$ *T w u' uBest lBest* . *fst* (*append-heuristic T w* (*uBest*,*lBest*) *u'*) $\in$ {*u'*,*uBest*}
shows $\forall$ *v* . *v* $\in$ *list.set X* $\longrightarrow$ $\neg$ *converge M1 u v* $\longrightarrow$ $\neg$ *converge M2 u v*
(**is** *?P1 X*)
and *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*))
(**is** *?P2 X*)
⟨*proof*⟩

lemma *spyh-distinguish-preserves-divergence* :
fixes *M1* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm*
assumes *observable M1*
and *observable M2*

and *minimal M1*
and *minimal M2*
and $u \in L\ M1$ **and** $u \in L\ M2$
and $\bigwedge \alpha\ \beta\ q1\ q2\ .\ q1 \in states\ M1 \Longrightarrow q2 \in states\ M1 \Longrightarrow q1 \neq q2 \Longrightarrow$ *distinguishes M1 q1 q2* (*get-distinguishing-trace q1 q2*)
and *convergence-graph-lookup-invar M1 M2 cg-lookup G*
and *convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*
and $list.set\ X \subseteq L\ M1$
and $list.set\ X \subseteq L\ M2$
and $L\ M1 \cap set$ (*fst* (*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*)) $= L\ M2 \cap set$ (*fst* (*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*))
and $\bigwedge T\ w\ u'\ uBest\ lBest\ .\ fst\ (append\text{-}heuristic\ T\ w\ (uBest,lBest)\ u') \in \{u',uBest\}$
and *preserves-divergence M1 M2* (*list.set X*)
shows *preserves-divergence M1 M2* (*Set.insert u* (*list.set X*))
(**is** *?P1 X*)
⟨*proof*⟩

21.3 HandleIOPair

definition *handle-io-pair* :: *bool* ⇒ *bool* ⇒ (('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒
('*a*,'*b*,'*c*) *state-cover-assignment* ⇒
('*b*×'*c*) *prefix-tree* ⇒
'*d* ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ '*d*) ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ ('*b*×'*c*) *list list*) ⇒
'*a* ⇒ '*b* ⇒ '*c* ⇒
(('*b*×'*c*) *prefix-tree* × '*d*)) **where**
handle-io-pair completeInputTraces useInputHeuristic M V T G cg-insert cg-lookup q x y =
distribute-extension M T G cg-lookup cg-insert (*V q*) [(*x*,*y*)] *completeInputTraces* (*if useInputHeuristic then append-heuristic-input M else append-heuristic-io*)

lemma *handle-io-pair-verifies-io-pair* : *verifies-io-pair* (*handle-io-pair b c*) *M1 M2 cg-lookup cg-insert*
⟨*proof*⟩

lemma *handle-io-pair-handles-io-pair* : *handles-io-pair* (*handle-io-pair b c*) *M1 M2 cg-lookup cg-insert*
⟨*proof*⟩

21.4 HandleStateCover

21.4.1 Dynamic

fun *handle-state-cover-dynamic* :: *bool* ⇒
bool ⇒

('a ⇒ 'a ⇒ ('b×'c) list) ⇒
('a::linorder,'b::linorder,'c::linorder) fsm ⇒
('a,'b,'c) state-cover-assignment ⇒
(('a,'b,'c) fsm ⇒ ('b×'c) prefix-tree ⇒ 'd) ⇒
('d ⇒ ('b×'c) list ⇒ 'd) ⇒
('d ⇒ ('b×'c) list ⇒ ('b×'c) list list) ⇒
(('b×'c) prefix-tree × 'd)

where

handle-state-cover-dynamic completeInputTraces useInputHeuristic get-distinguishing-trace M V cg-initial cg-insert cg-lookup =
(*let*
k = (2 ∗ *size M*);
heuristic = (*if useInputHeuristic then append-heuristic-input M else append-heuristic-io*);
rstates = *reachable-states-as-list M*;
T0' = *from-list* (*map V rstates*);
T0 = (*if completeInputTraces*
then Prefix-Tree.combine T0' (*from-list* (*concat* (*map* (λ *q* . *language-for-input M* (*initial M*) (*map fst* (*V q*))) *rstates*)))
else T0');
G0 = *cg-initial M T0*;
separate-state = (λ (*X*,*T*,*G*) *q* . *let u* = *V q*;
TG' = *spyh-distinguish M T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces heuristic*;
X' = *u*#*X*
in (*X'*,*TG'*))
in snd (*foldl separate-state* ([],*T0*,*G0*) *rstates*))

lemma *handle-state-cover-dynamic-separates-state-cover*:
fixes *M1* :: ('a::linorder,'b::linorder,'c::linorder) *fsm*
fixes *M2* :: ('e,'b,'c) *fsm*
fixes *cg-insert* :: ('d ⇒ ('b×'c) *list* ⇒ 'd)
assumes ⋀ α β *q1 q2* . *q1* ∈ *states M1* ⟹ *q2* ∈ *states M1* ⟹ *q1* ≠ *q2* ⟹ *distinguishes M1 q1 q2* (*dist-fun q1 q2*)
shows *separates-state-cover* (*handle-state-cover-dynamic b c dist-fun*) *M1 M2 cg-initial cg-insert cg-lookup*
⟨*proof*⟩

21.4.2 Static

fun *handle-state-cover-static* :: (*nat* ⇒ 'a ⇒ ('b×'c) *prefix-tree*) ⇒
('a::linorder,'b::linorder,'c::linorder) *fsm* ⇒
('a,'b,'c) *state-cover-assignment* ⇒
(('a,'b,'c) *fsm* ⇒ ('b×'c) *prefix-tree* ⇒ 'd) ⇒
('d ⇒ ('b×'c) *list* ⇒ 'd) ⇒
('d ⇒ ('b×'c) *list* ⇒ ('b×'c) *list list*) ⇒
(('b×'c) *prefix-tree* × 'd)

where

handle-state-cover-static dist-set M V cg-initial cg-insert cg-lookup =
(*let*
separate-state = (λ *T q* . *combine-after T* (*V q*) (*dist-set 0 q*));
T′ = *foldl separate-state empty* (*reachable-states-as-list M*);
G′ = *cg-initial M T′*
in (*T′*,*G′*))

lemma *handle-state-cover-static-applies-dist-sets*:
assumes *q* ∈ *reachable-states M1*
shows *set* (*dist-fun 0 q*) ⊆ *set* (*after* (*fst* (*handle-state-cover-static dist-fun M1 V cg-initial cg-insert cg-lookup*)) (*V q*))
(**is** *set* (*dist-fun 0 q*) ⊆ *set* (*after ?T* (*V q*)))
⟨*proof*⟩

lemma *handle-state-cover-static-separates-state-cover*:
fixes *M1* :: (*′a*::*linorder*,*′b*::*linorder*,*′c*::*linorder*) *fsm*
fixes *M2* :: (*′e*,*′b*,*′c*) *fsm*
fixes *cg-insert* :: (*′d* ⇒ (*′b*×*′c*) *list* ⇒ *′d*)
assumes *observable M1* ⟹ *minimal M1* ⟹ (⋀ *q1 q2* . *q1* ∈ *states M1* ⟹ *q2* ∈ *states M1* ⟹ *q1* ≠ *q2* ⟹ ∃ *io* . ∀ *k1 k2* . *io* ∈ *set* (*dist-fun k1 q1*) ∩ *set* (*dist-fun k2 q2*) ∧ *distinguishes M1 q1 q2 io*)
and ⋀ *k q* . *q* ∈ *states M1* ⟹ *finite-tree* (*dist-fun k q*)
shows *separates-state-cover* (*handle-state-cover-static dist-fun*) *M1 M2 cg-initial cg-insert cg-lookup*
⟨*proof*⟩

21.5 Establishing Convergence of Traces

21.5.1 Dynamic

fun *distinguish-from-set* :: (*′a*::*linorder*,*′b*::*linorder*,*′c*::*linorder*) *fsm* ⇒ (*′a*,*′b*,*′c*) *state-cover-assignment* ⇒ (*′b*×*′c*) *prefix-tree* ⇒ *′d* ⇒ (*′d* ⇒ (*′b*×*′c*) *list* ⇒ (*′b*×*′c*) *list list*) ⇒ (*′d* ⇒ (*′b*×*′c*) *list* ⇒ *′d*) ⇒ (*′a* ⇒ *′a* ⇒ (*′b*×*′c*) *list*) ⇒ (*′b*×*′c*) *list* ⇒ (*′b*×*′c*) *list* ⇒ (*′b*×*′c*) *list list* ⇒ *nat* ⇒ *nat* ⇒ *bool* ⇒ ((*′b*×*′c*) *prefix-tree* ⇒ (*′b*×*′c*) *list* ⇒ ((*′b*×*′c*) *list* × *int*) ⇒ (*′b*×*′c*) *list* ⇒ ((*′b*×*′c*) *list* × *int*)) ⇒ *bool* ⇒ ((*′b*×*′c*) *prefix-tree* × *′d*) **where**
distinguish-from-set M V T G cg-lookup cg-insert get-distinguishing-trace u v X k depth completeInputTraces append-heuristic u-is-v=
(*let TG′* = *spyh-distinguish M T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*;
vClass = *Set.insert v* (*list.set* (*cg-lookup* (*snd TG′*) *v*));
notReferenced = (¬ *u-is-v*) ∧ (∀ *q* ∈ *reachable-states M* . *V q* ∉ *vClass*);
TG″ = (*if notReferenced then spyh-distinguish M* (*fst TG′*) (*snd TG′*) *cg-lookup cg-insert get-distinguishing-trace v X k completeInputTraces append-heuristic*

else TG′)
in if depth > 0
then let X′ = if notReferenced then (v#u#X) else (u#X);
XY = List.product (inputs-as-list M) (outputs-as-list M);
handleIO = (λ (T,G) (x,y) . (let TGu = distribute-extension M T G cg-lookup cg-insert u [(x,y)] completeInputTraces append-heuristic;
TGv = if u-is-v then TGu else distribute-extension M (fst TGu) (snd TGu) cg-lookup cg-insert v [(x,y)] completeInputTraces append-heuristic
in if is-in-language M (initial M) (u@[(x,y)])
then distinguish-from-set M V (fst TGv) (snd TGv) cg-lookup cg-insert get-distinguishing-trace (u@[(x,y)]) (v@[(x,y)]) X′ k (depth − 1) completeInputTraces append-heuristic u-is-v
else TGv))
in foldl handleIO TG′′ XY
else TG′′)

lemma *distinguish-from-set-subset* :
set T ⊆ set (fst (distinguish-from-set M V T G cg-lookup cg-insert get-distinguishing-trace u v X k depth completeInputTraces append-heuristic u-is-v))
⟨*proof*⟩

lemma *distinguish-from-set-finite* :
fixes *T* :: (′b::linorder×′c::linorder) *prefix-tree*
assumes *finite-tree T*
shows *finite-tree (fst (distinguish-from-set M V T G cg-lookup cg-insert get-distinguishing-trace u v X k depth completeInputTraces append-heuristic u-is-v))*
⟨*proof*⟩

lemma *distinguish-from-set-properties* :
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *inputs M2 = inputs M1*
and *outputs M2 = outputs M1*
and *is-state-cover-assignment M1 V*
and *V ‘ reachable-states M1 ⊆ list.set X*
and *preserves-divergence M1 M2 (list.set X)*
and ⋀ *w . w ∈ list.set X ⟹ ∃ w′ . converge M1 w w′ ∧ converge M2 w w′*
and *converge M1 u v*
and *u ∈ L M2*
and *v ∈ L M2*
and *convergence-graph-lookup-invar M1 M2 cg-lookup G*
and *convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*
and ⋀ *α β q1 q2 . q1 ∈ states M1 ⟹ q2 ∈ states M1 ⟹ q1 ≠ q2 ⟹ distinguishes M1 q1 q2 (get-distinguishing-trace q1 q2)*

and *L M1* ∩ *set* (*fst* (*distinguish-from-set M1 V T G cg-lookup cg-insert get-distinguishing-trace u v X k depth completeInputTraces append-heuristic* (*u* = *v*))) = *L M2* ∩ *set* (*fst* (*distinguish-from-set M1 V T G cg-lookup cg-insert get-distinguishing-trace u v X k depth completeInputTraces append-heuristic* (*u* = *v*)))
and $\bigwedge$ *T w u′ uBest lBest* . *fst* (*append-heuristic T w* (*uBest*,*lBest*) *u′*) ∈ {*u′*,*uBest*}
shows ∀ *γ x y* . *length* (*γ*@[(*x*,*y*)]) ≤ *depth* ⟶
γ ∈ *LS M1* (*after-initial M1 u*) ⟶
x ∈ *inputs M1* ⟶ *y* ∈ *outputs M1* ⟶
L M1 ∩ (*list.set X* ∪ {*ω*@*ω′* | *ω ω′* . *ω* ∈ {*u*,*v*} ∧ *ω′* ∈ *list.set* (*prefixes* (*γ*@[(*x*,*y*)]))}) = *L M2* ∩ (*list.set X* ∪ {*ω*@*ω′* | *ω ω′* . *ω* ∈ {*u*,*v*} ∧ *ω′* ∈ *list.set* (*prefixes* (*γ*@[(*x*,*y*)]))})
∧ *preserves-divergence M1 M2* (*list.set X* ∪ {*ω*@*ω′* | *ω ω′* . *ω* ∈ {*u*,*v*} ∧ *ω′* ∈ *list.set* (*prefixes* (*γ*@[(*x*,*y*)]))})
(**is** *?P1a X u v depth*)
and *preserves-divergence M1 M2* (*list.set X* ∪ {*u*,*v*})
(**is** *?P1b X u v*)
and *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*distinguish-from-set M1 V T G cg-lookup cg-insert get-distinguishing-trace u v X k depth completeInputTraces append-heuristic* (*u* = *v*)))
(**is** *?P2 T G u v X depth*)
⟨*proof*⟩

lemma *distinguish-from-set-establishes-convergence* :
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
and *is-state-cover-assignment M1 V*
and *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)
and *L M1* ∩ (*V* ‘ *reachable-states M1*) = *L M2* ∩ *V* ‘ *reachable-states M1*
and *converge M1 u v*
and *u* ∈ *L M2*
and *v* ∈ *L M2*
and *convergence-graph-lookup-invar M1 M2 cg-lookup G*
and *convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*
and $\bigwedge$ *q1 q2* . *q1* ∈ *states M1* ⟹ *q2* ∈ *states M1* ⟹ *q1* ≠ *q2* ⟹ *distinguishes M1 q1 q2* (*get-distinguishing-trace q1 q2*)
and *L M1* ∩ *set* (*fst* (*distinguish-from-set M1 V T G cg-lookup cg-insert get-distinguishing-trace u v* (*map V* (*reachable-states-as-list M1*)) *k* (*m* − *size-r M1*) *completeInputTraces append-heuristic* (*u*=*v*))) = *L M2* ∩ *set* (*fst* (*distinguish-from-set M1 V T G cg-lookup cg-insert get-distinguishing-trace u v* (*map V* (*reachable-states-as-list M1*)) *k* (*m* − *size-r M1*) *completeInputTraces append-heuristic* (*u*=*v*)))
and $\bigwedge$ *T w u′ uBest lBest* . *fst* (*append-heuristic T w* (*uBest*,*lBest*) *u′*) ∈

{*u'*,*uBest*}
shows *converge M2 u v*
and *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*distinguish-from-set M1 V T G cg-lookup cg-insert get-distinguishing-trace u v* (*map V* (*reachable-states-as-list M1*)) *k* (*m* − *size-r M1*) *completeInputTraces append-heuristic* (*u*=*v*)))
⟨*proof*⟩

definition *establish-convergence-dynamic* :: *bool* ⇒ *bool* ⇒ ('*a* ⇒ '*a* ⇒ ('*b* × '*c*) *list*) ⇒
('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒
('*a*,'*b*,'*c*) *state-cover-assignment* ⇒
('*b*×'*c*) *prefix-tree* ⇒
'*d* ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ '*d*) ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ ('*b*×'*c*) *list list*) ⇒
nat ⇒
('*a*,'*b*,'*c*) *transition* ⇒
(('*b*×'*c*) *prefix-tree* × '*d*) **where**
establish-convergence-dynamic completeInputTraces useInputHeuristic dist-fun M1 V T G cg-insert cg-lookup m t =
distinguish-from-set M1 V T G cg-lookup cg-insert
dist-fun
((*V* (*t-source t*))@[(*t-input t*, *t-output t*)])
(*V* (*t-target t*))
(*map V* (*reachable-states-as-list M1*))
(*2* ∗ *size M1*)
(*m* − *size-r M1*)
completeInputTraces
(*if useInputHeuristic then append-heuristic-input M1 else append-heuristic-io*)
False

lemma *establish-convergence-dynamic-verifies-transition* :
assumes ⋀ *q1 q2* . *q1* ∈ *states M1* ⟹ *q2* ∈ *states M1* ⟹ *q1* ≠ *q2* ⟹ *distinguishes M1 q1 q2* (*dist-fun q1 q2*)
shows *verifies-transition* (*establish-convergence-dynamic b c dist-fun*) *M1 M2 V T0 cg-insert cg-lookup*
⟨*proof*⟩

definition *handleUT-dynamic* :: *bool* ⇒
bool ⇒
('*a* ⇒ '*a* ⇒ ('*b* × '*c*) *list*) ⇒
(('*a*,'*b*,'*c*) *fsm* ⇒ ('*a*,'*b*,'*c*) *state-cover-assignment* ⇒ ('*a*,'*b*,'*c*) *transition* ⇒ ('*a*,'*b*,'*c*) *transition list* ⇒ *nat* ⇒ *bool*) ⇒
('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒

```
                        ('a,'b,'c) state-cover-assignment ⇒
                        ('b×'c) prefix-tree ⇒
                        'd ⇒
                        ('d ⇒ ('b×'c) list ⇒ 'd) ⇒
                        ('d ⇒ ('b×'c) list ⇒ ('b×'c) list list) ⇒
                        ('d ⇒ ('b×'c) list ⇒ ('b×'c) list ⇒ 'd) ⇒
                        nat ⇒
                        ('a,'b,'c) transition ⇒
                        ('a,'b,'c) transition list ⇒
                        (('a,'b,'c) transition list × ('b×'c) prefix-tree × 'd)
where
handleUT-dynamic complete-input-traces
            use-input-heuristic
            dist-fun
            do-establish-convergence
            M
            V
            T
            G
            cg-insert
            cg-lookup
            cg-merge
            m
            t
            X
 =
 (let k          = (2 * size M);
      l          = (m − size-r M);
      heuristic = (if use-input-heuristic then append-heuristic-input M
                                        else append-heuristic-io);
      rstates   = (map V (reachable-states-as-list M));
      (T1,G1)   = handle-io-pair complete-input-traces
                          use-input-heuristic
                          M
                          V
                          T
                          G
                          cg-insert
                          cg-lookup
                          (t-source t)
                          (t-input t)
                          (t-output t);
      u          = ((V (t-source t))@[(t-input t, t-output t)]);
      v          = (V (t-target t));
      X'         = butlast X
   in if (do-establish-convergence M V t X' l)
      then let (T2,G2) = distinguish-from-set M
                                      V
                                      T1
```

G1
cg-lookup
cg-insert
dist-fun
u
v
rstates
k
l
complete-input-traces
heuristic
False;
G3 = *cg-merge G2 u v*
in
(*X′*,*T2*,*G3*)
else (*X′*,*distinguish-from-set M*
V
T1
G1
cg-lookup
cg-insert
dist-fun
u
u
rstates
k
l
complete-input-traces
heuristic
True))

lemma *handleUT-dynamic-handles-transition* :
fixes *M1*::(*′a*::*linorder*,*′b*::*linorder*,*′c*::*linorder*) *fsm*
fixes *M2*::(*′e*,*′b*,*′c*) *fsm*
assumes $\bigwedge$ *q1 q2* . *q1* ∈ *states M1* ⟹ *q2* ∈ *states M1* ⟹ *q1* ≠ *q2* ⟹ *distinguishes M1 q1 q2* (*dist-fun q1 q2*)
shows *handles-transition* (*handleUT-dynamic b c dist-fun d*) *M1 M2 V T0 cg-insert cg-lookup cg-merge*
⟨*proof*⟩

21.5.2 Static

fun *traces-to-check* :: (*′a*,*′b*::*linorder*,*′c*::*linorder*) *fsm* ⇒ *′a* ⇒ *nat* ⇒ (*′b*×*′c*) *list list* **where**
traces-to-check M q 0 = [] |
traces-to-check M q (*Suc k*) = (*let*
ios = *List.product* (*inputs-as-list M*) (*outputs-as-list M*)
in concat (*map* (λ(*x*,*y*) . *case h-obs M q x y of None* ⇒ [[(*x*,*y*)]] | *Some q′* ⇒

[(*x*,*y*)] # (*map* ((#) (*x*,*y*)) (*traces-to-check M q' k*))) *ios*))

lemma *traces-to-check-set* :
  fixes *M* :: ('*a*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
  assumes *observable M*
  and $q \in$ *states M*
shows *list.set* (*traces-to-check M q k*) = {(γ @ [(*x*, *y*)]) | γ *x y* . *length* (γ @ [(*x*, *y*)]) $\leq$ *k* $\wedge$ $\gamma \in$ *LS M q* $\wedge$ *x* $\in$ *inputs M* $\wedge$ *y* $\in$ *outputs M*}
  ⟨*proof*⟩

```
fun establish-convergence-static :: (nat ⇒ 'a ⇒ ('b×'c) prefix-tree) ⇒
                                    ('a::linorder,'b::linorder,'c::linorder) fsm ⇒
                                    ('a,'b,'c) state-cover-assignment ⇒
                                    ('b×'c) prefix-tree ⇒
                                    'd ⇒
                                    ('d ⇒ ('b×'c) list ⇒ 'd) ⇒
                                    ('d ⇒ ('b×'c) list ⇒ ('b×'c) list list) ⇒
                                    nat ⇒
                                    ('a,'b,'c) transition ⇒
                                    (('b×'c) prefix-tree × 'd)
  where
  establish-convergence-static dist-fun M V T G cg-insert cg-lookup m t =
    (let
        α = V (t-source t);
        xy = (t-input t, t-output t);
        β = V (t-target t);
        qSource = (after-initial M (V (t-source t)));
        qTarget = (after-initial M (V (t-target t)));
        k = m − size-r M;
        ttc = [] # traces-to-check M qTarget k;
        handleTrace = (λ (T,G) u .
          if is-in-language M qTarget u
            then let
                qu = FSM.after M qTarget u;
                ws = sorted-list-of-maximal-sequences-in-tree (dist-fun (Suc (length
u)) qu);
                appendDistTrace = (λ (T,G) w . let
                                                (T',G') = distribute-extension M T G
cg-lookup cg-insert α (xy#u@w) False (append-heuristic-input M)
                                              in distribute-extension M T' G' cg-lookup
cg-insert β (u@w) False (append-heuristic-input M))
                in foldl appendDistTrace (T,G) ws
            else let
                (T',G') = distribute-extension M T G cg-lookup cg-insert α (xy#u)
False (append-heuristic-input M)
                  in distribute-extension M T' G' cg-lookup cg-insert β u False
(append-heuristic-input M))
    in
      foldl handleTrace (T,G) ttc)
```

lemma *appendDistTrace-subset-helper* :
assumes *appendDistTrace* = (λ (*T*,*G*) *w* . *let*
(T',G') = *distribute-extension M T G cg-lookup cg-insert* α (*xy#u@w*) *False* (*append-heuristic-input M*)
in distribute-extension M T' G' *cg-lookup cg-insert* β (*u@w*) *False* (*append-heuristic-input M*))
shows *set T* $\subseteq$ *set* (*fst* (*appendDistTrace* (*T*,*G*) *w*))
⟨*proof*⟩

lemma *handleTrace-subset-helper* :
assumes *handleTrace* = (λ (*T*,*G*) *u* .
if is-in-language M qTarget u
then let
qu = *FSM.after M qTarget u*;
ws = *sorted-list-of-maximal-sequences-in-tree* (*dist-fun* (*Suc* (*length u*)) *qu*);
appendDistTrace = (λ (*T*,*G*) *w* . *let*
(T',G') = *distribute-extension M T G cg-lookup cg-insert* α (*xy#u@w*) *False* (*append-heuristic-input M*)
in distribute-extension M T' G' *cg-lookup cg-insert* β (*u@w*) *False* (*append-heuristic-input M*))
in foldl appendDistTrace (*T*,*G*) *ws*
else let
(T',G') = *distribute-extension M T G cg-lookup cg-insert* α (*xy#u*) *False* (*append-heuristic-input M*)
in distribute-extension M T' G' *cg-lookup cg-insert* β *u False* (*append-heuristic-input M*))
shows *set T* $\subseteq$ *set* (*fst* (*handleTrace* (*T*,*G*) *u*))
⟨*proof*⟩

lemma *establish-convergence-static-subset* :
set T $\subseteq$ *set* (*fst* (*establish-convergence-static dist-fun M V T G cg-insert cg-lookup m t*))
⟨*proof*⟩

lemma *establish-convergence-static-finite* :
fixes *M* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
assumes *finite-tree T*
shows *finite-tree* (*fst* (*establish-convergence-static dist-fun M V T G cg-insert cg-lookup m t*))
⟨*proof*⟩

lemma *establish-convergence-static-properties* :

assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
and $t \in$ *transitions M1*
and *t-source t* $\in$ *reachable-states M1*
and *is-state-cover-assignment M1 V*
and *V* (*t-source t*) @ [(*t-input t*, *t-output t*)] $\in$ *L M2*
and *V* ‘ *reachable-states M1* $\subseteq$ *set T*
and *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)
and *convergence-graph-lookup-invar M1 M2 cg-lookup G*
and *convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*
and $\bigwedge$ *q1 q2* . *q1* $\in$ *states M1* $\Longrightarrow$ *q2* $\in$ *states M1* $\Longrightarrow$ *q1* $\neq$ *q2* $\Longrightarrow$ $\exists$ *io* . $\forall$ *k1 k2* . *io* $\in$ *set* (*dist-fun k1 q1*) $\cap$ *set* (*dist-fun k2 q2*) $\wedge$ *distinguishes M1 q1 q2 io*
and $\bigwedge$ *q* . *q* $\in$ *reachable-states M1* $\Longrightarrow$ *set* (*dist-fun 0 q*) $\subseteq$ *set* (*after T* (*V q*))
and $\bigwedge$ *q k* . *q* $\in$ *states M1* $\Longrightarrow$ *finite-tree* (*dist-fun k q*)
and *L M1* $\cap$ *set* (*fst* (*establish-convergence-static dist-fun M1 V T G cg-insert cg-lookup m t*)) = *L M2* $\cap$ *set* (*fst* (*establish-convergence-static dist-fun M1 V T G cg-insert cg-lookup m t*))
shows $\forall$ γ *x y* . *length* (γ@[(*x*,*y*)]) $\leq$ *m* − *size-r M1* $\longrightarrow$
γ $\in$ *LS M1* (*after-initial M1* (*V* (*t-source t*) @ [(*t-input t*, *t-output t*)])) $\longrightarrow$
x $\in$ *inputs M1* $\longrightarrow$ *y* $\in$ *outputs M1* $\longrightarrow$
L M1 $\cap$ ((*V* ‘ *reachable-states M1*) $\cup$ {ω@ω' | ω ω' . ω $\in$ {((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]), (*V* (*t-target t*))} $\wedge$ ω' $\in$ *list.set* (*prefixes* (γ@[(*x*,*y*)]))}) = *L M2* $\cap$ ((*V* ‘ *reachable-states M1*) $\cup$ {ω@ω' | ω ω' . ω $\in$ {((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]), (*V* (*t-target t*))} $\wedge$ ω' $\in$ *list.set* (*prefixes* (γ@[(*x*,*y*)]))})
$\wedge$ *preserves-divergence M1 M2* ((*V* ‘ *reachable-states M1*) $\cup$ {ω@ω' | ω ω' . ω $\in$ {((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]), (*V* (*t-target t*))} $\wedge$ ω' $\in$ *list.set* (*prefixes* (γ@[(*x*,*y*)]))})
(**is** *?P1a*)
and *preserves-divergence M1 M2* ((*V* ‘ *reachable-states M1*) $\cup$ {((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]), (*V* (*t-target t*))})
(**is** *?P1b*)
and *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*establish-convergence-static dist-fun M1 V T G cg-insert cg-lookup m t*))
(**is** *?P2*)
⟨*proof*⟩

lemma *establish-convergence-static-establishes-convergence* :
assumes *observable M1*

and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
and *t* ∈ *transitions M1*
and *t-source t* ∈ *reachable-states M1*
and *is-state-cover-assignment M1 V*
and *V* (*t-source t*) @ [(*t-input t*, *t-output t*)] ∈ *L M2*
and *V* ' *reachable-states M1* ⊆ *set T*
and *preserves-divergence M1 M2* (*V* ' *reachable-states M1*)
and *convergence-graph-lookup-invar M1 M2 cg-lookup G*
and *convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*
and ⋀ *q1 q2* . *q1* ∈ *states M1* ⟹ *q2* ∈ *states M1* ⟹ *q1* ≠ *q2* ⟹ ∃ *io* . ∀ *k1 k2* . *io* ∈ *set* (*dist-fun k1 q1*) ∩ *set* (*dist-fun k2 q2*) ∧ *distinguishes M1 q1 q2 io*
and ⋀ *q* . *q* ∈ *reachable-states M1* ⟹ *set* (*dist-fun 0 q*) ⊆ *set* (*after T* (*V q*))
and ⋀ *q k* . *q* ∈ *states M1* ⟹ *finite-tree* (*dist-fun k q*)
and *L M1* ∩ *set* (*fst* (*establish-convergence-static dist-fun M1 V T G cg-insert cg-lookup m t*)) = *L M2* ∩ *set* (*fst* (*establish-convergence-static dist-fun M1 V T G cg-insert cg-lookup m t*))
shows *converge M2* (*V* (*t-source t*) @ [(*t-input t*, *t-output t*)]) (*V* (*t-target t*))
(**is** *converge M2 ?u ?v*)
⟨*proof*⟩

lemma *establish-convergence-static-verifies-transition* :
assumes ⋀ *q1 q2* . *q1* ∈ *states M1* ⟹ *q2* ∈ *states M1* ⟹ *q1* ≠ *q2* ⟹ ∃ *io* . ∀ *k1 k2* . *io* ∈ *set* (*dist-fun k1 q1*) ∩ *set* (*dist-fun k2 q2*) ∧ *distinguishes M1 q1 q2 io*
and ⋀ *q k* . *q* ∈ *states M1* ⟹ *finite-tree* (*dist-fun k q*)
shows *verifies-transition* (*establish-convergence-static dist-fun*) *M1 M2 V* (*fst* (*handle-state-cover-static dist-fun M1 V cg-initial cg-insert cg-lookup*)) *cg-insert cg-lookup*
⟨*proof*⟩

definition *handleUT-static* :: (*nat* ⇒ *'a* ⇒ (*'b*×*'c*) *prefix-tree*) ⇒
((*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* ⇒
(*'a*,*'b*,*'c*) *state-cover-assignment* ⇒
(*'b*×*'c*) *prefix-tree* ⇒
'd ⇒
(*'d* ⇒ (*'b*×*'c*) *list* ⇒ *'d*) ⇒
(*'d* ⇒ (*'b*×*'c*) *list* ⇒ (*'b*×*'c*) *list list*) ⇒

('d ⇒ ('b×'c) list ⇒ ('b×'c) list ⇒ 'd) ⇒
nat ⇒
('a,'b,'c) transition ⇒
('a,'b,'c) transition list ⇒
(('a,'b,'c) transition list × ('b×'c) prefix-tree × 'd))

where
handleUT-static dist-fun M V T G cg-insert cg-lookup cg-merge l t X = (*let*
(*T1*,*G1*) = *handle-io-pair False False M V T G cg-insert cg-lookup* (*t-source t*) (*t-input t*) (*t-output t*);
(*T2*,*G2*) = *establish-convergence-static dist-fun M V T1 G1 cg-insert cg-lookup l t*;
G3 = *cg-merge G2* ((*V* (*t-source t*))@[(*t-input t*, *t-output t*)]) (*V* (*t-target t*))
in (*X*,*T2*,*G3*))

lemma *handleUT-static-handles-transition* :
fixes *M1*::('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2*::('*e*,'*b*,'*c*) *fsm*
assumes ⋀ *q1 q2* . *q1* ∈ *states M1* ⟹ *q2* ∈ *states M1* ⟹ *q1* ≠ *q2* ⟹ ∃ *io* . ∀ *k1 k2* . *io* ∈ *set* (*dist-fun k1 q1*) ∩ *set* (*dist-fun k2 q2*) ∧ *distinguishes M1 q1 q2 io*
and ⋀ *q k* . *q* ∈ *states M1* ⟹ *finite-tree* (*dist-fun k q*)
shows *handles-transition* (*handleUT-static dist-fun*) *M1 M2 V* (*fst* (*handle-state-cover-static dist-fun M1 V cg-initial cg-insert cg-lookup*)) *cg-insert cg-lookup cg-merge*
⟨*proof*⟩

21.6 Distinguishing Traces

21.6.1 Symmetry

The following lemmata serve to show that the function to choose distinguishing sequences returns the same sequence for reversed pairs, thus ensuring that the HSIs do not contain two sequences for the same pair of states.

lemma *select-diverging-ofsm-table-io-sym* :
assumes *observable M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *ofsm-table M* (λ*q* . *states M*) (*Suc k*) *q1* ≠ *ofsm-table M* (λ*q* . *states M*) (*Suc k*) *q2*
assumes (*select-diverging-ofsm-table-io M q1 q2* (*Suc k*)) = (*io*,(*a*,*b*))
shows (*select-diverging-ofsm-table-io M q2 q1* (*Suc k*)) = (*io*,(*b*,*a*))
⟨*proof*⟩

lemma *assemble-distinguishing-sequence-from-ofsm-table-sym* :
assumes *observable M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*

and *ofsm-table M* (λq . *states M*) *k q1* $\neq$ *ofsm-table M* (λq . *states M*) *k q2*
shows *assemble-distinguishing-sequence-from-ofsm-table M q1 q2 k* = *assemble-distinguishing-sequence-from-of*
M q2 q1 k
⟨*proof*⟩

lemma *find-first-distinct-ofsm-table-sym* :
assumes *q1* $\in$ *FSM.states M*
and *q2* $\in$ *FSM.states M*
and *ofsm-table-fix M* (λq . *states M*) *0 q1* $\neq$ *ofsm-table-fix M* (λq . *states M*) *0 q2*
shows *find-first-distinct-ofsm-table M q1 q2* = *find-first-distinct-ofsm-table M q2 q1*
⟨*proof*⟩

lemma *get-distinguishing-sequence-from-ofsm-tables-sym* :
assumes *observable M*
and *minimal M*
and *q1* $\in$ *states M*
and *q2* $\in$ *states M*
and *q1* $\neq$ *q2*
shows *get-distinguishing-sequence-from-ofsm-tables M q1 q2* = *get-distinguishing-sequence-from-ofsm-tables M q2 q1*
⟨*proof*⟩

21.6.2 Harmonised State Identifiers

fun *add-distinguishing-sequence* :: ('*a*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ (('*b*×'*c*) *list* × '*a*) × (('*b*×'*c*) *list* × '*a*) ⇒ ('*b*×'*c*) *prefix-tree* ⇒ ('*b*×'*c*) *prefix-tree* **where**
add-distinguishing-sequence M ((α,*q1*), (β,*q2*)) *t* = *insert empty* (*get-distinguishing-sequence-from-ofsm-tables*
M q1 q2)

lemma *add-distinguishing-sequence-distinguishes* :
assumes *observable M*
and *minimal M*
and $\alpha \in L\ M$
and $\beta \in L\ M$
and *after-initial M* α $\neq$ *after-initial M* β
shows ∃ *io* ∈ *set* (*add-distinguishing-sequence M* ((α,*after-initial M* α),(β,*after-initial M* β)) *t*) ∪ (*set* (*after t* α) ∩ *set* (*after t* β)) . *distinguishes M* (*after-initial M* α) (*after-initial M* β) *io*
⟨*proof*⟩

lemma *add-distinguishing-sequence-finite* :
finite-tree (*add-distinguishing-sequence M* ((α,*after-initial M* α),(β,*after-initial M* β)) *t*)
⟨*proof*⟩

fun *get-HSI* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ '*a* ⇒ ('*b*×'*c*) *prefix-tree*
where
get-HSI M q = *from-list* (*map* (λ*q'* . *get-distinguishing-sequence-from-ofsm-tables M q q'*) (*filter* ((≠) *q*) (*states-as-list M*)))

lemma *get-HSI-elem* :
assumes *q2* ∈ *states M*
and *q2* ≠ *q1*
shows *get-distinguishing-sequence-from-ofsm-tables M q1 q2* ∈ *set* (*get-HSI M q1*)
⟨*proof*⟩

lemma *get-HSI-distinguishes* :
assumes *observable M*
and *minimal M*
and *q1* ∈ *states M* **and** *q2* ∈ *states M* **and** *q1* ≠ *q2*
shows ∃ *io* ∈ *set* (*get-HSI M q1*) ∩ *set* (*get-HSI M q2*) . *distinguishes M q1 q2 io*
⟨*proof*⟩

lemma *get-HSI-finite* :
finite-tree (*get-HSI M q*)
⟨*proof*⟩

21.6.3 Distinguishing Sets

fun *distinguishing-set* :: ('*a* :: *linorder*, '*b* :: *linorder*, '*c* :: *linorder*) *fsm* ⇒ ('*b* × '*c*) *prefix-tree* **where**
distinguishing-set M = (*let*
pairs = *filter* (λ (*x*,*y*) . *x* ≠ *y*) (*list-ordered-pairs* (*states-as-list M*))
in from-list (*map* (*case-prod* (*get-distinguishing-sequence-from-ofsm-tables M*)) *pairs*))

lemma *distinguishing-set-distinguishes* :
assumes *observable M*
and *minimal M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *q1* ≠ *q2*
shows ∃ *io* ∈ *set* (*distinguishing-set M*) . *distinguishes M q1 q2 io*
⟨*proof*⟩

lemma *distinguishing-set-finite* :
finite-tree (*distinguishing-set M*)
⟨*proof*⟩

function (*domintros*) *intersection-is-distinguishing* :: ('*a*,'*b*,'*c*) *fsm* ⇒ ('*b* × '*c*) *prefix-tree* ⇒ '*a* ⇒ ('*b* × '*c*) *prefix-tree* ⇒ '*a* ⇒ *bool* **where**
  intersection-is-distinguishing M (*PT t1*) *q1* (*PT t2*) *q2* =
    (∃ (*x*,*y*) ∈ *dom t1* ∩ *dom t2* .
      case h-obs M q1 x y of
        None ⇒ *h-obs M q2 x y* ≠ *None* |
        Some q1' ⇒ (*case h-obs M q2 x y of*
          None ⇒ *True* |
          Some q2' ⇒ *intersection-is-distinguishing M* (*the* (*t1* (*x*,*y*))) *q1'* (*the* (*t2* (*x*,*y*))) *q2'*))
  ⟨*proof*⟩
termination
⟨*proof*⟩

lemma *intersection-is-distinguishing-code*[*code*] :
  intersection-is-distinguishing M (*MPT t1*) *q1* (*MPT t2*) *q2* =
    (∃ (*x*,*y*) ∈ *Mapping.keys t1* ∩ *Mapping.keys t2* .
      case h-obs M q1 x y of
        None ⇒ *h-obs M q2 x y* ≠ *None* |
        Some q1' ⇒ (*case h-obs M q2 x y of*
          None ⇒ *True* |
            Some q2' ⇒ *intersection-is-distinguishing M* (*the* (*Mapping.lookup t1* (*x*,*y*))) *q1'* (*the* (*Mapping.lookup t2* (*x*,*y*))) *q2'*))
  ⟨*proof*⟩

lemma *intersection-is-distinguishing-correctness* :
  assumes *observable M*
  and *q1* ∈ *states M*
  and *q2* ∈ *states M*
shows *intersection-is-distinguishing M t1 q1 t2 q2* = (∃ *io* . *isin t1 io* ∧ *isin t2 io* ∧ *distinguishes M q1 q2 io*)
  (**is** *?P1* = *?P2*)
⟨*proof*⟩

fun *contains-distinguishing-trace* :: ('*a*,'*b*,'*c*) *fsm* ⇒ ('*b* × '*c*) *prefix-tree* ⇒ '*a* ⇒ '*a* ⇒ *bool* **where**
  contains-distinguishing-trace M T q1 q2 = *intersection-is-distinguishing M T q1 T q2*

lemma *contains-distinguishing-trace-code*[*code*] :
  contains-distinguishing-trace M (*MPT t1*) *q1 q2* =
    (∃ (*x*,*y*) ∈ *Mapping.keys t1*.

case h-obs M q1 x y of
None ⇒ *h-obs M q2 x y* ≠ *None* |
Some q1′ ⇒ (*case h-obs M q2 x y of*
None ⇒ *True* |
Some q2′ ⇒ *contains-distinguishing-trace M* (*the* (*Mapping.lookup t1* (*x*,*y*))) *q1′ q2′*))
⟨*proof*⟩

lemma *contains-distinguishing-trace-correctness* :
assumes *observable M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *contains-distinguishing-trace M t q1 q2* = (∃ *io* . *isin t io* ∧ *distinguishes M q1 q2 io*)
⟨*proof*⟩

fun *distinguishing-set-reduced* :: (′*a* :: *linorder*, ′*b* :: *linorder*, ′*c* :: *linorder*) *fsm* ⇒ (′*b* × ′*c*) *prefix-tree* **where**
distinguishing-set-reduced M = (*let*
pairs = *filter* (λ (*q*,*q*′) . *q* ≠ *q*′) (*list-ordered-pairs* (*states-as-list M*));
handlePair = (λ *W* (*q*,*q*′) . *if contains-distinguishing-trace M W q q*′
then W
else insert W (*get-distinguishing-sequence-from-ofsm-tables M q q*′))
in foldl handlePair empty pairs)

lemma *distinguishing-set-reduced-distinguishes* :
assumes *observable M*
and *minimal M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *q1* ≠ *q2*
shows ∃ *io* ∈ *set* (*distinguishing-set-reduced M*) . *distinguishes M q1 q2 io*
⟨*proof*⟩

lemma *distinguishing-set-reduced-finite* :
finite-tree (*distinguishing-set-reduced M*)
⟨*proof*⟩

fun *add-distinguishing-set* :: (′*a* :: *linorder*, ′*b* :: *linorder*, ′*c* :: *linorder*) *fsm* ⇒ ((′*b*×′*c*) *list* × ′*a*) × ((′*b*×′*c*) *list* × ′*a*) ⇒ (′*b*×′*c*) *prefix-tree* ⇒ (′*b*×′*c*) *prefix-tree* **where**
add-distinguishing-set M - t = *distinguishing-set M*

lemma *add-distinguishing-set-distinguishes* :
assumes *observable M*
and *minimal M*
and $\alpha \in L\ M$
and $\beta \in L\ M$
and *after-initial M* $\alpha \neq$ *after-initial M* β
shows $\exists$ *io* $\in$ *set* (*add-distinguishing-set M* ((α,*after-initial M* α),(β,*after-initial M* β)) *t*) $\cup$ (*set* (*after t* α) $\cap$ *set* (*after t* β)) . *distinguishes M* (*after-initial M* α) (*after-initial M* β) *io*
⟨*proof*⟩

lemma *add-distinguishing-set-finite* :
finite-tree ((*add-distinguishing-set M*) *x t*)
⟨*proof*⟩

21.7 Transition Sorting

definition *sort-unverified-transitions-by-state-cover-length* :: ('*a* :: *linorder*,'*b* :: *linorder*,'*c* :: *linorder*) *fsm* ⇒ ('*a*,'*b*,'*c*) *state-cover-assignment* ⇒ ('*a*,'*b*,'*c*) *transition list* ⇒ ('*a*,'*b*,'*c*) *transition list* **where**
sort-unverified-transitions-by-state-cover-length M V ts = (*let*
default-weight = *2* ∗ *size M*;
weights = *mapping-of* (*map* (λ*t* . (*t*, *length* (*V* (*t-source t*)) + *length* (*V* (*t-target t*)))) *ts*);
weight = (λ*q* . *case Mapping.lookup weights q of Some w* ⇒ *w* | *None* ⇒ *default-weight*)
in mergesort-by-rel (λ *t1 t2* . *weight t1* ≤ *weight t2*) *ts*)

lemma *sort-unverified-transitions-by-state-cover-length-retains-set* :
List.set xs = *List.set* (*sort-unverified-transitions-by-state-cover-length M1* (*get-state-cover M1*) *xs*)
⟨*proof*⟩

end

22 Test Suites for Language Equivalence

This file introduces a type for test suites represented as a prefix tree in which each IO-pair is additionally labeled by a boolean value representing whether the IO-pair should be exhibited by the SUT in order to pass the test suite.

theory *Test-Suite-Representations*
imports ../*Minimisation* ../*Prefix-Tree*
begin

type-synonym ('*b*,'*c*) *test-suite* = (('*b* × '*c*) × *bool*) *prefix-tree*

function (*domintros*) *test-suite-from-io-tree* :: (*'a,'b,'c*) *fsm* ⇒ *'a* ⇒ (*'b* × *'c*) *prefix-tree* ⇒ (*'b,'c*) *test-suite* **where**
test-suite-from-io-tree M q (*PT m*) = *PT* (λ ((*x*,*y*),*b*) . *case m* (*x*,*y*) *of*
None ⇒ *None* |
Some t ⇒ (*case h-obs M q x y of*
None ⇒ (*if b then None else Some empty*) |
Some q' ⇒ (*if b then Some* (*test-suite-from-io-tree M q' t*) *else None*)))
⟨*proof*⟩
termination
⟨*proof*⟩

22.1 Transforming an IO-prefix-tree to a test suite

lemma *test-suite-from-io-tree-set* :
assumes *observable M*
and *q* ∈ *states M*
shows (*set* (*test-suite-from-io-tree M q t*)) = ((λ *xs* . *map* (λ *x* . (*x*,*True*)) *xs*) ' (*set t* ∩ *LS M q*))
∪ ((λ *xs* . (*map* (λ *x* . (*x*,*True*)) (*butlast xs*))@[(*last xs*,*False*)]) ' {*xs*@[*x*] | *xs x* . *xs* ∈ *set t* ∩ *LS M q* ∧ *xs*@[*x*] ∈ *set t* − *LS M q*})
(**is** *?S1 q t* = *?S2 q t*)
⟨*proof*⟩

function (*domintros*) *passes-test-suite* :: (*'a,'b,'c*) *fsm* ⇒ *'a* ⇒ (*'b,'c*) *test-suite* ⇒ *bool* **where**
passes-test-suite M q (*PT m*) = (∀ *xyb* ∈ *dom m* . *case h-obs M q* (*fst* (*fst xyb*)) (*snd* (*fst xyb*)) *of*
None ⇒ ¬(*snd xyb*) |
Some q' ⇒ *snd xyb* ∧ *passes-test-suite M q'* (*case m xyb of Some t* ⇒ *t*))
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *passes-test-suite-iff* :
assumes *observable M*
and *q* ∈ *states M*
shows *passes-test-suite M q t* = (∀ *iob* ∈ *set t* . (*map fst iob*) ∈ *LS M q* ⟷ *list-all snd iob*)
⟨*proof*⟩

lemma *passes-test-suite-from-io-tree* :
assumes *observable M*

and *observable I*
and *qM ∈ states M*
and *qI ∈ states I*
shows *passes-test-suite I qI (test-suite-from-io-tree M qM t) = ((set t ∩ LS M qM) = (set t ∩ LS I qI))*
⟨*proof*⟩

22.2 Code Refinement

context includes *lifting-syntax*
begin

lemma *map-entries-parametric*:
((A ===> B) ===> (A ===> C ===> rel-option D) ===> (B ===> rel-option C) ===> A ===> rel-option D)
(λf g m x. case (m ∘ f) x of None ⇒ None | Some y ⇒ g x y) (λf g m x. case (m ∘ f) x of None ⇒ None | Some y ⇒ g x y)
⟨*proof*⟩

end

lift-definition *map-entries* :: *('c ⇒ 'a) ⇒ ('c ⇒ 'b ⇒ 'd option) ⇒ ('a, 'b) mapping ⇒ ('c, 'd) mapping*
is *λf g m x. case (m ∘ f) x of None ⇒ None | Some y ⇒ g x y* **parametric** *map-entries-parametric* ⟨*proof*⟩

lemma *test-suite-from-io-tree-MPT[code]* :
test-suite-from-io-tree M q (MPT m) =
MPT (map-entries
fst
(λ ((x,y),b) t . (case h-obs M q x y of
None ⇒ (if b then None else Some empty) |
Some q' ⇒ (if b then Some (test-suite-from-io-tree M q' t) else None)))
m)
(**is** *?t M q (MPT m) = MPT (?f M q m)*)
⟨*proof*⟩

lemma *passes-test-suite-MPT[code]*:
passes-test-suite M q (MPT m) = (∀ xyb ∈ Mapping.keys m . case h-obs M q (fst (fst xyb)) (snd (fst xyb)) of
None ⇒ ¬(snd xyb) |
Some q' ⇒ snd xyb ∧ passes-test-suite M q' (case Mapping.lookup m xyb of Some t ⇒ t))
⟨*proof*⟩

22.3 Pass relations on list of lists representations of test suites

fun *passes-test-case* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ *'a* $\Rightarrow$ $(('b \times 'c) \times$ *bool*) *list* $\Rightarrow$ *bool* **where**
passes-test-case M q [] = *True* |
passes-test-case M q $(((x,y),b)\#io)$ = (*if b*
then case h-obs M q x y of
Some q' $\Rightarrow$ *passes-test-case M q' io* |
None $\Rightarrow$ *False*
else h-obs M q x y = *None*)

lemma *passes-test-case-iff* :
assumes *observable M*
and $q \in$ *states M*
shows *passes-test-case M q iob* = ((*map fst* (*takeWhile snd iob*) $\in$ *LS M q*)
$\wedge$ ($\neg$ (*list-all snd iob*) $\longrightarrow$ *map fst* (*take* (*Suc* (*length* (*takeWhile snd iob*))) *iob*) $\notin$ *LS M q*))
⟨*proof*⟩

lemma *test-suite-from-io-tree-finite-tree* :
assumes *observable M*
and *qM* $\in$ *states M*
and *finite-tree t*
shows *finite-tree* (*test-suite-from-io-tree M qM t*)
⟨*proof*⟩

lemma *passes-test-case-prefix* :
assumes *observable M*
and *passes-test-case M q* (*iob*@*iob'*)
shows *passes-test-case M q iob*
⟨*proof*⟩

lemma *passes-test-cases-of-test-suite* :
assumes *observable M*
and *observable I*
and *qM* $\in$ *states M*
and *qI* $\in$ *states I*
and *finite-tree t*
shows *list-all* (*passes-test-case I qI*) (*sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M qM t*)) = *passes-test-suite I qI* (*test-suite-from-io-tree M qM t*)
(**is** *?P1* = *?P2*)
⟨*proof*⟩

lemma *passes-test-cases-from-io-tree* :
assumes *observable M*

and *observable I*
and *qM* ∈ *states M*
and *qI* ∈ *states I*
and *finite-tree t*
shows *list-all* (*passes-test-case I qI*) (*sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M qM t*)) = ((*set t* ∩ *LS M qM*) = (*set t* ∩ *LS I qI*))
⟨*proof*⟩

22.4 Alternative Representations

22.4.1 Pass and Fail Traces

type-synonym ('*b*,'*c*) *pass-traces* = ('*b* × '*c*) *list list*
type-synonym ('*b*,'*c*) *fail-traces* = ('*b* × '*c*) *list list*
type-synonym ('*b*,'*c*) *trace-test-suite* = ('*b*,'*c*) *pass-traces* × ('*b*,'*c*) *fail-traces*

fun *trace-test-suite-from-tree* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ ('*b* × '*c*) *prefix-tree* ⇒ ('*b*,'*c*) *trace-test-suite* **where**
trace-test-suite-from-tree M T = (*let*
(*passes'*,*fails*) = *separate-by* (*is-in-language M* (*initial M*)) (*sorted-list-of-sequences-in-tree T*);
passes = *sorted-list-of-maximal-sequences-in-tree* (*from-list passes'*)
in (*passes*, *fails*))

lemma *trace-test-suite-from-tree-language-equivalence* :
assumes *observable M* **and** *finite-tree T*
shows (*L M* ∩ *set T* = *L M'* ∩ *set T*) = (*list.set* (*fst* (*trace-test-suite-from-tree M T*)) ⊆ *L M'* ∧ *L M'* ∩ *list.set* (*snd* (*trace-test-suite-from-tree M T*)) = {})
⟨*proof*⟩

22.4.2 Input Sequences

fun *test-suite-to-input-sequences* :: ('*b*::*linorder*×'*c*::*linorder*) *prefix-tree* ⇒ '*b list list* **where**
test-suite-to-input-sequences T = *sorted-list-of-maximal-sequences-in-tree* (*from-list* (*map input-portion* (*sorted-list-of-maximal-sequences-in-tree T*)))

lemma *test-suite-to-input-sequences-pass* :
fixes *T* :: ('*b*::*linorder* × '*c*::*linorder*) *prefix-tree*
assumes *finite-tree T*
and (*L M* = *L M'*) ⟷ (*L M* ∩ *set T* = *L M'* ∩ *set T*)
shows (*L M* = *L M'*) ⟷ ({*io* ∈ *L M* . (∃ *xs* ∈ *list.set* (*test-suite-to-input-sequences T*) . ∃ *xs'* ∈ *list.set* (*prefixes xs*) . *input-portion io* = *xs'*)}
= {*io* ∈ *L M'* . (∃ *xs* ∈ *list.set* (*test-suite-to-input-sequences T*) . ∃ *xs'* ∈ *list.set* (*prefixes xs*) . *input-portion io* = *xs'*)})
⟨*proof*⟩

lemma *test-suite-to-input-sequences-pass-alt-def* :
fixes *T* :: ('*b*::*linorder* × '*c*::*linorder*) *prefix-tree*

assumes *finite-tree T*
and $(L\ M = L\ M') \longleftrightarrow (L\ M \cap set\ T = L\ M' \cap set\ T)$
shows $(L\ M = L\ M') \longleftrightarrow (\forall\ xs \in list.set\ (test\text{-}suite\text{-}to\text{-}input\text{-}sequences\ T)\ .\ \forall\ xs' \in list.set\ (prefixes\ xs)\ .\ \{io \in L\ M\ .\ input\text{-}portion\ io = xs'\} = \{io \in L\ M'\ .\ input\text{-}portion\ io = xs'\})$
⟨*proof*⟩

end

23 Simple Convergence Graphs

This theory introduces a very simple implementation of convergence graphs that consists of a list of convergent classes represented as sets of traces.

theory *Simple-Convergence-Graph*
imports *Convergence-Graph*
begin

23.1 Basic Definitions

type-synonym *'a simple-cg = 'a list fset list*

definition *simple-cg-empty* :: *'a simple-cg* **where**
simple-cg-empty = []

fun *simple-cg-lookup* :: (*'a::linorder*) *simple-cg* ⇒ *'a list* ⇒ *'a list list* **where**
simple-cg-lookup xs ys = *sorted-list-of-fset* (*finsert ys* (*foldl* (|∪|) *fempty* (*filter* (λ*x* . *ys* |∈| *x*) *xs*)))

fun *simple-cg-lookup-with-conv* :: (*'a::linorder*) *simple-cg* ⇒ *'a list* ⇒ *'a list list* **where**
simple-cg-lookup-with-conv g ys = (*let*
lookup-for-prefix = (λ*i* . *let*
pref = *take i ys*;
suff = *drop i ys*;
pref-conv = (*foldl* (|∪|) *fempty* (*filter* (λ*x* . *pref* |∈| *x*) *g*))
in fimage (λ *pref'* . *pref'*@*suff*) *pref-conv*)
in sorted-list-of-fset (*finsert ys* (*foldl* (λ *cs i* . *lookup-for-prefix i* |∪| *cs*) *fempty* [*0*..<*Suc* (*length ys*)])))

fun *simple-cg-insert'* :: (*'a::linorder*) *simple-cg* ⇒ *'a list* ⇒ *'a simple-cg* **where**
simple-cg-insert' xs ys = (*case find* (λ*x* . *ys* |∈| *x*) *xs*
of Some x ⇒ *xs* |
None ⇒ {|*ys*|}#*xs*)

fun *simple-cg-insert* :: ('*a*::*linorder*) *simple-cg* ⇒ '*a list* ⇒ '*a simple-cg* **where**
simple-cg-insert xs ys = *foldl* (λ *xs' ys'* . *simple-cg-insert' xs' ys'*) *xs* (*prefixes ys*)

fun *simple-cg-initial* :: ('*a*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ ('*b*×'*c*) *prefix-tree* ⇒ ('*b*×'*c*) *simple-cg* **where**
simple-cg-initial M1 T = *foldl* (λ *xs' ys'* . *simple-cg-insert' xs' ys'*) *simple-cg-empty* (*filter* (*is-in-language M1* (*initial M1*)) (*sorted-list-of-sequences-in-tree T*))

23.2 Merging by Closure

The following implementation of the merge operation follows the closure operation described by Simão et al. in Simão, A., Petrenko, A. and Yevtushenko, N. (2012), On reducing test length for FSMs with extra states. Softw. Test. Verif. Reliab., 22: 435-454. https://doi.org/10.1002/stvr.452. That is, two traces u and v are merged by adding u,v to the list of convergent classes followed by computing the closure of the graph based on two operations: (1) classes A and B can be merged if there exists some class C such that C contains some w1, w2 and there exists some w such that A contains w1.w and B contains w2.w. (2) classes A and B can be merged if one is a subset of the other.

fun *can-merge-by-suffix* :: '*a list fset* ⇒ '*a list fset* ⇒ '*a list fset* ⇒ *bool* **where**
can-merge-by-suffix x x1 x2 = (∃ α β γ . α |∈| *x* ∧ β |∈| *x* ∧ α@γ |∈| *x1* ∧ β@γ |∈| *x2*)

lemma *can-merge-by-suffix-code*[*code*] :
can-merge-by-suffix x x1 x2 =
(∃ *ys* ∈ *fset x* .
∃ *ys1* ∈ *fset x1* .
is-prefix ys ys1 ∧
(∃ *ys'* ∈ *fset x* . *ys'*@(*drop* (*length ys*) *ys1*) |∈| *x2*))
(**is** *?P1* = *?P2*)
⟨*proof*⟩

fun *prefixes-in-list-helper* :: '*a* ⇒ '*a list list* ⇒ (*bool* × '*a list list*) ⇒ *bool* × '*a list list* **where**
prefixes-in-list-helper x [] *res* = *res* |
prefixes-in-list-helper x ([]#*yss*) *res* = *prefixes-in-list-helper x yss* (*True*, *snd res*) |
prefixes-in-list-helper x ((*y*#*ys*)#*yss*) *res* =
(*if x* = *y then prefixes-in-list-helper x yss* (*fst res*, *ys* # *snd res*)
else prefixes-in-list-helper x yss res)

fun *prefixes-in-list* :: '*a list* ⇒ '*a list* ⇒ '*a list list* ⇒ '*a list list* ⇒ '*a list list* **where**
prefixes-in-list [] *prev yss res* = (*if List.member yss* [] *then prev*#*res else res*) |
prefixes-in-list (*x*#*xs*) *prev yss res* = (*let*
(*b*,*yss'*) = *prefixes-in-list-helper x yss* (*False*,[])

in if b then prefixes-in-list xs (prev@[x]) yss' (prev # res)
else prefixes-in-list xs (prev@[x]) yss' res)

fun *prefixes-in-set* :: (*'a::linorder*) *list* ⇒ *'a list fset* ⇒ *'a list list* **where**
prefixes-in-set xs yss = *prefixes-in-list xs* [] (*sorted-list-of-fset yss*) []

value *prefixes-in-list* [*1::nat,2,3,4,5*] []
[[*1,2,3*], [*1,2,4*], [*1,3*], [], [*1*], [*1,5,3*], [*2,5*]] []

value *prefixes-in-list-helper* (*1::nat*)
[[*1,2,3*], [*1,2,4*], [*1,3*], [], [*1*], [*1,5,3*], [*2,5*]]
(*False*,[])

lemma *prefixes-in-list-helper-prop* :
shows *fst* (*prefixes-in-list-helper x yss res*) = (*fst res* ∨ [] ∈ *list.set yss*) (**is** *?P1*)
and *list.set* (*snd* (*prefixes-in-list-helper x yss res*)) = *list.set* (*snd res*) ∪ {*ys* . *x#ys* ∈ *list.set yss*} (**is** *?P2*)
⟨*proof*⟩

lemma *prefixes-in-list-prop* :
shows *list.set* (*prefixes-in-list xs prev yss res*) = *list.set res* ∪ {*prev@ys* | *ys* . *ys* ∈ *list.set* (*prefixes xs*) ∧ *ys* ∈ *list.set yss*}
⟨*proof*⟩

lemma *prefixes-in-set-prop* :
list.set (*prefixes-in-set xs yss*) = *list.set* (*prefixes xs*) ∩ *fset yss*
⟨*proof*⟩

lemma *can-merge-by-suffix-validity* :
assumes *observable M1* **and** *observable M2*
and ⋀ *u v* . *u* |∈| *x* ⟹ *v* |∈| *x* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and ⋀ *u v* . *u* |∈| *x1* ⟹ *v* |∈| *x1* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and ⋀ *u v* . *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and *can-merge-by-suffix x x1 x2*
and *u* |∈| (*x1* |∪| *x2*)
and *v* |∈| (*x1* |∪| *x2*)
and *u* ∈ *L M1* **and** *u* ∈ *L M2*
shows *converge M1 u v* ∧ *converge M2 u v*
⟨*proof*⟩

fun *simple-cg-closure-phase-1-helper'* :: *'a list fset* ⇒ *'a list fset* ⇒ *'a simple-cg* ⇒ (*bool* × *'a list fset* × *'a simple-cg*) **where**
simple-cg-closure-phase-1-helper' x x1 xs =
(*let* (*x2s*,*others*) = *separate-by* (*can-merge-by-suffix x x1*) *xs*;
x1Union = *foldl* (|∪|) *x1 x2s*
in (*x2s* ≠ [],*x1Union*,*others*))

lemma *simple-cg-closure-phase-1-helper'-False* :
¬*fst* (*simple-cg-closure-phase-1-helper' x x1 xs*) ⟹ *simple-cg-closure-phase-1-helper' x x1 xs* = (*False*,*x1*,*xs*)
⟨*proof*⟩

lemma *simple-cg-closure-phase-1-helper'-True* :
assumes *fst* (*simple-cg-closure-phase-1-helper' x x1 xs*)
shows *length* (*snd* (*snd* (*simple-cg-closure-phase-1-helper' x x1 xs*))) < *length xs*
⟨*proof*⟩

lemma *simple-cg-closure-phase-1-helper'-length* :
length (*snd* (*snd* (*simple-cg-closure-phase-1-helper' x x1 xs*))) ≤ *length xs*
⟨*proof*⟩

lemma *simple-cg-closure-phase-1-helper'-validity-fst* :
assumes *observable M1* **and** *observable M2*
and ⋀ *u v* . *u* |∈| *x* ⟹ *v* |∈| *x* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and ⋀ *u v* . *u* |∈| *x1* ⟹ *v* |∈| *x1* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and ⋀ *x2 u v* . *x2* ∈ *list.set xs* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and *u* |∈| *fst* (*snd* (*simple-cg-closure-phase-1-helper' x x1 xs*))
and *v* |∈| *fst* (*snd* (*simple-cg-closure-phase-1-helper' x x1 xs*))
and *u* ∈ *L M1* **and** *u* ∈ *L M2*
shows *converge M1 u v* ∧ *converge M2 u v*
⟨*proof*⟩

lemma *simple-cg-closure-phase-1-helper'-validity-snd* :
assumes ⋀ *x2 u v* . *x2* ∈ *list.set xs* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and *x2* ∈ *list.set* (*snd* (*snd* (*simple-cg-closure-phase-1-helper' x x1 xs*)))
and *u* |∈| *x2*
and *v* |∈| *x2*
and *u* ∈ *L M1* **and** *u* ∈ *L M2*
shows *converge M1 u v* ∧ *converge M2 u v*
⟨*proof*⟩

fun *simple-cg-closure-phase-1-helper* :: *'a list fset* ⇒ *'a simple-cg* ⇒ (*bool* × *'a*

simple-cg) ⇒ (*bool* × *'a simple-cg*) **where**
simple-cg-closure-phase-1-helper x [] (*b*,*done*) = (*b*,*done*) |
simple-cg-closure-phase-1-helper x (*x1*#*xs*) (*b*,*done*) = (*let* (*hasChanged*,*x1'*,*xs'*) = *simple-cg-closure-phase-1-helper' x x1 xs*
in simple-cg-closure-phase-1-helper x xs' (*b* ∨ *hasChanged*, *x1'* # *done*))

lemma *simple-cg-closure-phase-1-helper-validity* :
assumes *observable M1* **and** *observable M2*
and ⋀ *u v* . *u* |∈| *x* ⟹ *v* |∈| *x* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and ⋀ *x2 u v* . *x2* ∈ *list.set don* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and ⋀ *x2 u v* . *x2* ∈ *list.set xss* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and *x2* ∈ *list.set* (*snd* (*simple-cg-closure-phase-1-helper x xss* (*b*,*don*)))
and *u* |∈| *x2*
and *v* |∈| *x2*
and *u* ∈ *L M1* **and** *u* ∈ *L M2*
shows *converge M1 u v* ∧ *converge M2 u v*
⟨*proof*⟩

lemma *simple-cg-closure-phase-1-helper-length* :
length (*snd* (*simple-cg-closure-phase-1-helper x xss* (*b*,*don*))) ≤ *length xss* + *length don*
⟨*proof*⟩

lemma *simple-cg-closure-phase-1-helper-True* :
assumes *fst* (*simple-cg-closure-phase-1-helper x xss* (*False*,*don*))
and *xss* ≠ []
shows *length* (*snd* (*simple-cg-closure-phase-1-helper x xss* (*False*,*don*))) < *length xss* + *length don*
⟨*proof*⟩

fun *simple-cg-closure-phase-1* :: *'a simple-cg* ⇒ (*bool* × *'a simple-cg*) **where**
simple-cg-closure-phase-1 xs = *foldl* (λ (*b*,*xs*) *x*. *let* (*b'*,*xs'*) = *simple-cg-closure-phase-1-helper x xs* (*False*,[]) *in* (*b*∨*b'*,*xs'*)) (*False*,*xs*) *xs*

lemma *simple-cg-closure-phase-1-validity* :
assumes *observable M1* **and** *observable M2*
and ⋀ *x2 u v* . *x2* ∈ *list.set xs* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*

and *x2* ∈ *list.set* (*snd* (*simple-cg-closure-phase-1 xs*))
and *u* |∈| *x2*
and *v* |∈| *x2*
and *u* ∈ *L M1* **and** *u* ∈ *L M2*
shows *converge M1 u v* ∧ *converge M2 u v*
⟨*proof*⟩

lemma *simple-cg-closure-phase-1-length-helper* :
length (*snd* (*foldl* (λ (*b*,*xs*) *x* . *let* (*b'*,*xs'*) = *simple-cg-closure-phase-1-helper x xs* (*False*,[]) *in* (*b*∨*b'*,*xs'*)) (*False*,*xs*) *xss*)) ≤ *length xs*
⟨*proof*⟩

lemma *simple-cg-closure-phase-1-length* :
length (*snd* (*simple-cg-closure-phase-1 xs*)) ≤ *length xs*
⟨*proof*⟩

lemma *simple-cg-closure-phase-1-True* :
assumes *fst* (*simple-cg-closure-phase-1 xs*)
shows *length* (*snd* (*simple-cg-closure-phase-1 xs*)) < *length xs*
⟨*proof*⟩

fun *can-merge-by-intersection* :: *'a list fset* ⇒ *'a list fset* ⇒ *bool* **where**
can-merge-by-intersection x1 x2 = (∃ α . α |∈| *x1* ∧ α |∈| *x2*)

lemma *can-merge-by-intersection-code*[*code*] :
can-merge-by-intersection x1 x2 = (∃ α ∈ *fset x1* . α |∈| *x2*)
⟨*proof*⟩

lemma *can-merge-by-intersection-validity* :
assumes ⋀ *u v* . *u* |∈| *x1* ⟹ *v* |∈| *x1* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and ⋀ *u v* . *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and *can-merge-by-intersection x1 x2*
and *u* |∈| (*x1* |∪| *x2*)
and *v* |∈| (*x1* |∪| *x2*)
and *u* ∈ *L M1*
and *u* ∈ *L M2*
shows *converge M1 u v* ∧ *converge M2 u v*
⟨*proof*⟩

fun *simple-cg-closure-phase-2-helper* :: *'a list fset* ⇒ *'a simple-cg* ⇒ (*bool* × *'a list fset* × *'a simple-cg*) **where**
simple-cg-closure-phase-2-helper x1 xs =

(*let* (*x2s*,*others*) = *separate-by* (*can-merge-by-intersection x1*) *xs*;
x1Union = *foldl* (|∪|) *x1 x2s*
in (*x2s* ≠ [],*x1Union*,*others*))

lemma *simple-cg-closure-phase-2-helper-length* :
length (*snd* (*snd* (*simple-cg-closure-phase-2-helper x1 xs*))) ≤ *length xs*
⟨*proof*⟩

lemma *simple-cg-closure-phase-2-helper-validity-fst* :
assumes ⋀ *u v* . *u* |∈| *x1* ⟹ *v* |∈| *x1* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and ⋀ *x2 u v* . *x2* ∈ *list.set xs* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and *u* |∈| *fst* (*snd* (*simple-cg-closure-phase-2-helper x1 xs*))
and *v* |∈| *fst* (*snd* (*simple-cg-closure-phase-2-helper x1 xs*))
and *u* ∈ *L M1*
and *u* ∈ *L M2*
shows *converge M1 u v* ∧ *converge M2 u v*
⟨*proof*⟩

lemma *simple-cg-closure-phase-2-helper-validity-snd* :
assumes ⋀ *x2 u v* . *x2* ∈ *list.set xs* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and *x2* ∈ *list.set* (*snd* (*snd* (*simple-cg-closure-phase-2-helper x1 xs*)))
and *u* |∈| *x2*
and *v* |∈| *x2*
and *u* ∈ *L M1*
and *u* ∈ *L M2*
shows *converge M1 u v* ∧ *converge M2 u v*
⟨*proof*⟩

lemma *simple-cg-closure-phase-2-helper-True* :
assumes *fst* (*simple-cg-closure-phase-2-helper x xs*)
shows *length* (*snd* (*snd* (*simple-cg-closure-phase-2-helper x xs*))) < *length xs*
⟨*proof*⟩

function *simple-cg-closure-phase-2′* :: ′*a simple-cg* ⇒ (*bool* × ′*a simple-cg*) ⇒ (*bool* × ′*a simple-cg*) **where**
simple-cg-closure-phase-2′ [] (*b*,*done*) = (*b*,*done*) |
simple-cg-closure-phase-2′ (*x*#*xs*) (*b*,*done*) = (*let* (*hasChanged*,*x′*,*xs′*) = *simple-cg-closure-phase-2-helper x xs*
in if hasChanged then simple-cg-closure-phase-2′ xs′ (*True*,*x′*#*done*)
else simple-cg-closure-phase-2′ xs (*b*,*x*#*done*))
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *simple-cg-closure-phase-2′-validity* :
assumes $\bigwedge$ *x2 u v* . *x2* ∈ *list.set don* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and $\bigwedge$ *x2 u v* . *x2* ∈ *list.set xss* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and *x2* ∈ *list.set* (*snd* (*simple-cg-closure-phase-2′ xss* (*b*,*don*)))
and *u* |∈| *x2*
and *v* |∈| *x2*
and *u* ∈ *L M1*
and *u* ∈ *L M2*
shows *converge M1 u v* ∧ *converge M2 u v*
⟨*proof*⟩

lemma *simple-cg-closure-phase-2′-length* :
length (*snd* (*simple-cg-closure-phase-2′ xss* (*b*,*don*))) ≤ *length xss* + *length don*
⟨*proof*⟩

lemma *simple-cg-closure-phase-2′-True* :
assumes *fst* (*simple-cg-closure-phase-2′ xss* (*False*,*don*))
and *xss* ≠ []
shows *length* (*snd* (*simple-cg-closure-phase-2′ xss* (*False*,*don*))) < *length xss* + *length don*
⟨*proof*⟩

fun *simple-cg-closure-phase-2* :: ′*a simple-cg* ⇒ (*bool* × ′*a simple-cg*) **where**
simple-cg-closure-phase-2 xs = *simple-cg-closure-phase-2′ xs* (*False*,[])

lemma *simple-cg-closure-phase-2-validity* :
assumes $\bigwedge$ *x2 u v* . *x2* ∈ *list.set xss* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and *x2* ∈ *list.set* (*snd* (*simple-cg-closure-phase-2 xss*))
and *u* |∈| *x2*
and *v* |∈| *x2*
and *u* ∈ *L M1*
and *u* ∈ *L M2*
shows *converge M1 u v* ∧ *converge M2 u v*
⟨*proof*⟩

lemma *simple-cg-closure-phase-2-length* :
length (*snd* (*simple-cg-closure-phase-2 xss*)) ≤ *length xss*
⟨*proof*⟩

lemma *simple-cg-closure-phase-2-True* :

assumes *fst* (*simple-cg-closure-phase-2 xss*)
shows *length* (*snd* (*simple-cg-closure-phase-2 xss*)) < *length xss*
⟨*proof*⟩

function *simple-cg-closure* :: *'a simple-cg* ⇒ *'a simple-cg* **where**
simple-cg-closure g = (*let* (*hasChanged1*,*g1*) = *simple-cg-closure-phase-1 g*;
(*hasChanged2*,*g2*) = *simple-cg-closure-phase-2 g1*
in if hasChanged1 ∨ *hasChanged2*
then simple-cg-closure g2
else g2)
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *simple-cg-closure-validity* :
assumes *observable M1* **and** *observable M2*
and ⋀ *x2 u v* . *x2* ∈ *list.set g* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and *x2* ∈ *list.set* (*simple-cg-closure g*)
and *u* |∈| *x2*
and *v* |∈| *x2*
and *u* ∈ *L M1*
and *u* ∈ *L M2*
shows *converge M1 u v* ∧ *converge M2 u v*
⟨*proof*⟩

fun *simple-cg-insert-with-conv* :: (*'a*::*linorder*) *simple-cg* ⇒ *'a list* ⇒ *'a simple-cg*
where
simple-cg-insert-with-conv g ys = (*let*
insert-for-prefix = (λ *g i* . *let*
pref = *take i ys*;
suff = *drop i ys*;
pref-conv = *simple-cg-lookup g pref*
in foldl (λ *g' ys'* . *simple-cg-insert' g'* (*ys'*@*suff*)) *g pref-conv*);
g' = *simple-cg-insert g ys*;
g'' = *foldl insert-for-prefix g'* [*0*..<*length ys*]
in simple-cg-closure g'')

fun *simple-cg-merge* :: *'a simple-cg* ⇒ *'a list* ⇒ *'a list* ⇒ *'a simple-cg* **where**
simple-cg-merge g ys1 ys2 = *simple-cg-closure* ({|*ys1*,*ys2*|}#*g*)

lemma *simple-cg-merge-validity* :

assumes *observable M1* **and** *observable M2*
and $converge\ M1\ u'\ v' \wedge converge\ M2\ u'\ v'$
and $\bigwedge x2\ u\ v\ .\ x2 \in list.set\ g \Longrightarrow u\ |\in|\ x2 \Longrightarrow v\ |\in|\ x2 \Longrightarrow u \in L\ M1 \Longrightarrow u \in L\ M2 \Longrightarrow converge\ M1\ u\ v \wedge converge\ M2\ u\ v$
and $x2 \in list.set\ (simple\text{-}cg\text{-}merge\ g\ u'\ v')$
and $u\ |\in|\ x2$
and $v\ |\in|\ x2$
and $u \in L\ M1$
and $u \in L\ M2$
shows $converge\ M1\ u\ v \wedge converge\ M2\ u\ v$
⟨*proof*⟩

23.3 Invariants

lemma *simple-cg-lookup-iff* :
$\beta \in list.set\ (simple\text{-}cg\text{-}lookup\ G\ \alpha) \longleftrightarrow (\beta = \alpha \vee (\exists\ x\ .\ x \in list.set\ G \wedge \alpha\ |\in|\ x \wedge \beta\ |\in|\ x))$
⟨*proof*⟩

lemma *simple-cg-insert′-invar* :
convergence-graph-insert-invar M1 M2 simple-cg-lookup simple-cg-insert′
⟨*proof*⟩

lemma *simple-cg-insert′-foldl-helper*:
assumes $list.set\ xss \subseteq L\ M1 \cap L\ M2$
and $(\bigwedge\alpha\ \beta.\ \beta \in list.set\ (simple\text{-}cg\text{-}lookup\ G\ \alpha) \Longrightarrow \alpha \in L\ M1 \Longrightarrow \alpha \in L\ M2 \Longrightarrow converge\ M1\ \alpha\ \beta \wedge converge\ M2\ \alpha\ \beta)$
shows $(\bigwedge\alpha\ \beta.\ \beta \in list.set\ (simple\text{-}cg\text{-}lookup\ (foldl\ (\lambda\ xs'\ ys'\ .\ simple\text{-}cg\text{-}insert'\ xs'\ ys')\ G\ xss)\ \alpha) \Longrightarrow \alpha \in L\ M1 \Longrightarrow \alpha \in L\ M2 \Longrightarrow converge\ M1\ \alpha\ \beta \wedge converge\ M2\ \alpha\ \beta)$
⟨*proof*⟩

lemma *simple-cg-insert-invar* :
convergence-graph-insert-invar M1 M2 simple-cg-lookup simple-cg-insert
⟨*proof*⟩

lemma *simple-cg-closure-invar-helper* :
assumes *observable M1* **and** *observable M2*
and $(\bigwedge\alpha\ \beta.\ \beta \in list.set\ (simple\text{-}cg\text{-}lookup\ G\ \alpha) \Longrightarrow \alpha \in L\ M1 \Longrightarrow \alpha \in L\ M2 \Longrightarrow converge\ M1\ \alpha\ \beta \wedge converge\ M2\ \alpha\ \beta)$
and $\beta \in list.set\ (simple\text{-}cg\text{-}lookup\ (simple\text{-}cg\text{-}closure\ G)\ \alpha)$
and $\alpha \in L\ M1$ **and** $\alpha \in L\ M2$
shows $converge\ M1\ \alpha\ \beta \wedge converge\ M2\ \alpha\ \beta$
⟨*proof*⟩

lemma *simple-cg-merge-invar* :
assumes *observable M1* **and** *observable M2*
shows *convergence-graph-merge-invar M1 M2 simple-cg-lookup simple-cg-merge*
⟨*proof*⟩

lemma *simple-cg-empty-invar* :
convergence-graph-lookup-invar M1 M2 simple-cg-lookup simple-cg-empty
⟨*proof*⟩

lemma *simple-cg-initial-invar* :
assumes *observable M1*
shows *convergence-graph-initial-invar M1 M2 simple-cg-lookup simple-cg-initial*
⟨*proof*⟩

lemma *simple-cg-insert-with-conv-invar* :
assumes *observable M1*
assumes *observable M2*
shows *convergence-graph-insert-invar M1 M2 simple-cg-lookup simple-cg-insert-with-conv*
⟨*proof*⟩

lemma *simple-cg-lookup-with-conv-from-lookup-invar*:
assumes *observable M1* **and** *observable M2*
and *convergence-graph-lookup-invar M1 M2 simple-cg-lookup G*
shows *convergence-graph-lookup-invar M1 M2 simple-cg-lookup-with-conv G*
⟨*proof*⟩

lemma *simple-cg-lookup-from-lookup-invar-with-conv*:
assumes *convergence-graph-lookup-invar M1 M2 simple-cg-lookup-with-conv G*
shows *convergence-graph-lookup-invar M1 M2 simple-cg-lookup G*
⟨*proof*⟩

lemma *simple-cg-lookup-invar-with-conv-eq* :
assumes *observable M1* **and** *observable M2*
shows *convergence-graph-lookup-invar M1 M2 simple-cg-lookup-with-conv G* =
convergence-graph-lookup-invar M1 M2 simple-cg-lookup G
⟨*proof*⟩

lemma *simple-cg-insert-invar-with-conv* :
assumes *observable M1* **and** *observable M2*
shows *convergence-graph-insert-invar M1 M2 simple-cg-lookup-with-conv simple-cg-insert*

⟨*proof*⟩

lemma *simple-cg-merge-invar-with-conv* :
assumes *observable M1* **and** *observable M2*
shows *convergence-graph-merge-invar M1 M2 simple-cg-lookup-with-conv simple-cg-merge*
⟨*proof*⟩

lemma *simple-cg-initial-invar-with-conv* :
assumes *observable M1* **and** *observable M2*
shows *convergence-graph-initial-invar M1 M2 simple-cg-lookup-with-conv simple-cg-initial*
⟨*proof*⟩

end

24 Intermediate Frameworks

This theory provides partial applications of the H, SPY, and Pair-Frameworks.

theory *Intermediate-Frameworks*
imports *Intermediate-Implementations Test-Suite-Representations ../OFSM-Tables-Refined Simple-Convergence-Graph Empty-Convergence-Graph*
begin

24.1 Partial Applications of the SPY-Framework

definition *spy-framework-static-with-simple-graph* :: ($'a$::*linorder*,$'b$::*linorder*,$'c$::*linorder*) *fsm* ⇒
(*nat* ⇒ $'a$ ⇒ ($'b \times 'c$) *prefix-tree*) ⇒
nat ⇒
($'b \times 'c$) *prefix-tree*
where
spy-framework-static-with-simple-graph M1
dist-fun
m
= *spy-framework M1*
get-state-cover-assignment
(*handle-state-cover-static dist-fun*)
(λ *M V ts* . *ts*)
(*establish-convergence-static dist-fun*)
(*handle-io-pair False True*)
simple-cg-initial
simple-cg-insert
simple-cg-lookup-with-conv
simple-cg-merge
m

lemma *spy-framework-static-with-simple-graph-completeness-and-finiteness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*d*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
and ⋀ *q1 q2* . *q1* ∈ *states M1* ⟹ *q2* ∈ *states M1* ⟹ *q1* ≠ *q2* ⟹ ∃ *io* . ∀ *k1 k2* . *io* ∈ *set* (*dist-fun k1 q1*) ∩ *set* (*dist-fun k2 q2*) ∧ *distinguishes M1 q1 q2 io*
and ⋀ *q k* . *q* ∈ *states M1* ⟹ *finite-tree* (*dist-fun k q*)
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*spy-framework-static-with-simple-graph M1 dist-fun m*)) = (*L M2* ∩ *set* (*spy-framework-static-with-simple-graph M1 dist-fun m*)))
and *finite-tree* (*spy-framework-static-with-simple-graph M1 dist-fun m*)
⟨*proof*⟩

definition *spy-framework-static-with-empty-graph* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒
(*nat* ⇒ '*a* ⇒ ('*b*×'*c*) *prefix-tree*) ⇒
nat ⇒
('*b*×'*c*) *prefix-tree*
where
spy-framework-static-with-empty-graph M1
dist-fun
m
= *spy-framework M1*
get-state-cover-assignment
(*handle-state-cover-static dist-fun*)
(λ *M V ts* . *ts*)
(*establish-convergence-static dist-fun*)
(*handle-io-pair False True*)
empty-cg-initial
empty-cg-insert
empty-cg-lookup
empty-cg-merge
m

lemma *spy-framework-static-with-empty-graph-completeness-and-finiteness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*d*,'*b*,'*c*) *fsm*

assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* $\leq$ *m*
and *size M2* $\leq$ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
and $\bigwedge$ *q1 q2 . q1* $\in$ *states M1* $\Longrightarrow$ *q2* $\in$ *states M1* $\Longrightarrow$ *q1* $\neq$ *q2* $\Longrightarrow$ $\exists$ *io* . $\forall$ *k1 k2 . io* $\in$ *set* (*dist-fun k1 q1*) $\cap$ *set* (*dist-fun k2 q2*) $\wedge$ *distinguishes M1 q1 q2 io*
and $\bigwedge$ *q k . q* $\in$ *states M1* $\Longrightarrow$ *finite-tree* (*dist-fun k q*)
shows (*L M1* = *L M2*) $\longleftrightarrow$ ((*L M1* $\cap$ *set* (*spy-framework-static-with-empty-graph M1 dist-fun m*)) = (*L M2* $\cap$ *set* (*spy-framework-static-with-empty-graph M1 dist-fun m*)))
and *finite-tree* (*spy-framework-static-with-empty-graph M1 dist-fun m*)
⟨*proof*⟩

24.2 Partial Applications of the H-Framework

definition *h-framework-static-with-simple-graph* :: (*'a::linorder,'b::linorder,'c::linorder*) *fsm* ⇒
(*nat* ⇒ *'a* ⇒ (*'b*×*'c*) *prefix-tree*) ⇒
nat ⇒
(*'b*×*'c*) *prefix-tree*
where
h-framework-static-with-simple-graph M1 dist-fun m =
h-framework M1
get-state-cover-assignment
(*handle-state-cover-static dist-fun*)
(λ *M V ts . ts*)
(*handleUT-static dist-fun*)
(*handle-io-pair False False*)
simple-cg-initial
simple-cg-insert
simple-cg-lookup-with-conv
simple-cg-merge
m

lemma *h-framework-static-with-simple-graph-completeness-and-finiteness* :
fixes *M1* :: (*'a::linorder,'b::linorder,'c::linorder*) *fsm*
fixes *M2* :: (*'e,'b,'c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* $\leq$ *m*
and *size M2* $\leq$ *m*
and *inputs M2* = *inputs M1*

and *outputs M2 = outputs M1*
and $\bigwedge$ *q1 q2 . q1* ∈ *states M1* ⟹ *q2* ∈ *states M1* ⟹ *q1* ≠ *q2* ⟹ ∃ *io .* ∀ *k1 k2 . io* ∈ *set* (*dist-fun k1 q1*) ∩ *set* (*dist-fun k2 q2*) ∧ *distinguishes M1 q1 q2 io*
and $\bigwedge$ *q k . q* ∈ *states M1* ⟹ *finite-tree* (*dist-fun k q*)
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*h-framework-static-with-simple-graph M1 dist-fun m*)) = (*L M2* ∩ *set* (*h-framework-static-with-simple-graph M1 dist-fun m*)))
and *finite-tree* (*h-framework-static-with-simple-graph M1 dist-fun m*)
⟨*proof*⟩

definition *h-framework-static-with-simple-graph-lists* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ (*nat* ⇒ '*a* ⇒ ('*b*×'*c*) *prefix-tree*) ⇒ *nat* ⇒ (('*b*×'*c*) × *bool*) *list list* **where**
h-framework-static-with-simple-graph-lists M dist-fun m = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*h-framework-static-with-simple-graph M dist-fun m*))

lemma *h-framework-static-with-simple-graph-lists-completeness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*d*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2 = inputs M1*
and *outputs M2 = outputs M1*
and $\bigwedge$ *q1 q2 . q1* ∈ *states M1* ⟹ *q2* ∈ *states M1* ⟹ *q1* ≠ *q2* ⟹ ∃ *io .* ∀ *k1 k2 . io* ∈ *set* (*dist-fun k1 q1*) ∩ *set* (*dist-fun k2 q2*) ∧ *distinguishes M1 q1 q2 io*
and $\bigwedge$ *q k . q* ∈ *states M1* ⟹ *finite-tree* (*dist-fun k q*)
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*initial M2*)) (*h-framework-static-with-simple-graph-li M1 dist-fun m*)
⟨*proof*⟩

definition *h-framework-static-with-empty-graph* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒
(*nat* ⇒ '*a* ⇒ ('*b*×'*c*) *prefix-tree*) ⇒
nat ⇒
('*b*×'*c*) *prefix-tree*
where
h-framework-static-with-empty-graph M1 dist-fun m =
h-framework M1
get-state-cover-assignment
(*handle-state-cover-static dist-fun*)
(λ *M V ts . ts*)
(*handleUT-static dist-fun*)

(*handle-io-pair False False*)
empty-cg-initial
empty-cg-insert
empty-cg-lookup
empty-cg-merge
m

lemma *h-framework-static-with-empty-graph-completeness-and-finiteness* :
fixes *M1* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm*
fixes *M2* :: (*'e*,*'b*,*'c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
and ⋀ *q1 q2* . *q1* ∈ *states M1* ⟹ *q2* ∈ *states M1* ⟹ *q1* ≠ *q2* ⟹ ∃ *io* . ∀ *k1 k2* . *io* ∈ *set* (*dist-fun k1 q1*) ∩ *set* (*dist-fun k2 q2*) ∧ *distinguishes M1 q1 q2 io*
and ⋀ *q k* . *q* ∈ *states M1* ⟹ *finite-tree* (*dist-fun k q*)
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*h-framework-static-with-empty-graph M1 dist-fun m*)) = (*L M2* ∩ *set* (*h-framework-static-with-empty-graph M1 dist-fun m*)))
and *finite-tree* (*h-framework-static-with-empty-graph M1 dist-fun m*)
⟨*proof*⟩

definition *h-framework-static-with-empty-graph-lists* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* ⇒ (*nat* ⇒ *'a* ⇒ (*'b*×*'c*) *prefix-tree*) ⇒ *nat* ⇒ ((*'b*×*'c*) × *bool*) *list list* **where**
h-framework-static-with-empty-graph-lists M dist-fun m = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*h-framework-static-with-empty-graph M dist-fun m*))

lemma *h-framework-static-with-empty-graph-lists-completeness* :
fixes *M1* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm*
fixes *M2* :: (*'d*,*'b*,*'c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
and ⋀ *q1 q2* . *q1* ∈ *states M1* ⟹ *q2* ∈ *states M1* ⟹ *q1* ≠ *q2* ⟹ ∃ *io* . ∀ *k1 k2* . *io* ∈ *set* (*dist-fun k1 q1*) ∩ *set* (*dist-fun k2 q2*) ∧ *distinguishes M1 q1 q2 io*
and ⋀ *q k* . *q* ∈ *states M1* ⟹ *finite-tree* (*dist-fun k q*)

shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*initial M2*)) (*h-framework-static-with-empty-graph-li*
M1 dist-fun m)
⟨*proof*⟩

definition *h-framework-dynamic* ::
(('*a*,'*b*,'*c*) *fsm* ⇒ ('*a*,'*b*,'*c*) *state-cover-assignment* ⇒ ('*a*,'*b*,'*c*) *transition* ⇒ ('*a*,'*b*,'*c*) *transition list* ⇒ *nat* ⇒ *bool*) ⇒
('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒
nat ⇒
bool ⇒
bool ⇒
('*b*×'*c*) *prefix-tree*
where
h-framework-dynamic convergence-decisision M1 m completeInputTraces useInputHeuristic =
h-framework M1
get-state-cover-assignment
(*handle-state-cover-dynamic completeInputTraces useInputHeuristic* (*get-distinguishing-sequence-from-ofsm-tables M1*))
sort-unverified-transitions-by-state-cover-length
(*handleUT-dynamic completeInputTraces useInputHeuristic* (*get-distinguishing-sequence-from-ofsm-tables M1*) *convergence-decisision*)
(*handle-io-pair completeInputTraces useInputHeuristic*)
simple-cg-initial
simple-cg-insert
simple-cg-lookup-with-conv
simple-cg-merge
m

lemma *h-framework-dynamic-completeness-and-finiteness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*e*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*h-framework-dynamic convergenceDecision M1 m completeInputTraces useInputHeuristic*)) = (*L M2* ∩ *set* (*h-framework-dynamic convergenceDecision M1 m completeInputTraces useInputHeuristic*)))
and *finite-tree* (*h-framework-dynamic convergenceDecision M1 m completeInputTraces useInputHeuristic*)
⟨*proof*⟩

definition *h-framework-dynamic-lists* :: (('*a*,'*b*,'*c*) *fsm* ⇒ ('*a*,'*b*,'*c*) *state-cover-assignment* ⇒ ('*a*,'*b*,'*c*) *transition* ⇒ ('*a*,'*b*,'*c*) *transition list* ⇒ *nat* ⇒ *bool*) ⇒ ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ *nat* ⇒ *bool* ⇒ *bool* ⇒ (('*b*×'*c*) × *bool*) *list list* **where**
h-framework-dynamic-lists convergenceDecision M m completeInputTraces useInputHeuristic = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*h-framework-dynamic convergenceDecision M m completeInputTraces useInputHeuristic*))

lemma *h-framework-dynamic-lists-completeness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*d*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*initial M2*)) (*h-framework-dynamic-lists convergenceDecision M1 m completeInputTraces useInputHeuristic*)
⟨*proof*⟩

24.3 Partial Applications of the Pair-Framework

definition *pair-framework-h-components* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ *nat* ⇒
(('*a*,'*b*,'*c*) *fsm* ⇒ ((('*b* × '*c*) *list* × '*a*) × ('*b* × '*c*) *list* × '*a* ⇒ ('*b* × '*c*) *prefix-tree* ⇒ ('*b* × '*c*) *prefix-tree*) ⇒
('*b*×'*c*) *prefix-tree*
where
pair-framework-h-components M m get-separating-traces = (*let*
V = *get-state-cover-assignment M*
in pair-framework M m (*get-initial-test-suite-H V*) (*get-pairs-H V*) *get-separating-traces*)

lemma *pair-framework-h-components-completeness-and-finiteness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*e*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
and $\bigwedge \alpha\ \beta\ t\ .\ \alpha \in L\ M1 \Longrightarrow \beta \in L\ M1 \Longrightarrow$ *after-initial M1* α ≠ *after-initial M1*

$\beta \Longrightarrow \exists$ *io* $\in$ *set* (*get-separating-traces M1* ((α,*after-initial M1* α),(β,*after-initial M1* β)) *t*) $\cup$ (*set* (*after t* α) $\cap$ *set* (*after t* β)) . *distinguishes M1* (*after-initial M1* α) (*after-initial M1* β) *io*
and $\bigwedge \alpha\ \beta\ t$. $\alpha \in L$ *M1* $\Longrightarrow \beta \in L$ *M1* $\Longrightarrow$ *after-initial M1* $\alpha \neq$ *after-initial M1* $\beta \Longrightarrow$ *finite-tree* (*get-separating-traces M1* ((α,*after-initial M1* α),(β,*after-initial M1* β)) *t*)
shows (*L M1* = *L M2*) $\longleftrightarrow$ ((*L M1* $\cap$ *set* (*pair-framework-h-components M1 m get-separating-traces*)) = (*L M2* $\cap$ *set* (*pair-framework-h-components M1 m get-separating-traces*)))
and *finite-tree* (*pair-framework-h-components M1 m get-separating-traces*)
⟨*proof*⟩

definition *pair-framework-h-components-2* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ *nat* ⇒
(('*a*,'*b*,'*c*) *fsm* ⇒ (('*b* × '*c*) *list* × '*a*) × ('*b* × '*c*) *list* × '*a* ⇒ ('*b* × '*c*) *prefix-tree* ⇒ ('*b* × '*c*) *prefix-tree*) ⇒
bool ⇒
('*b*×'*c*) *prefix-tree*
where
pair-framework-h-components-2 M m get-separating-traces c = (*let*
V = *get-state-cover-assignment M*
in pair-framework M m (*get-initial-test-suite-H-2 c V*) (*get-pairs-H V*) *get-separating-traces*)

lemma *pair-framework-h-components-2-completeness-and-finiteness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*e*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
and $\bigwedge \alpha\ \beta\ t$. $\alpha \in L$ *M1* $\Longrightarrow \beta \in L$ *M1* $\Longrightarrow$ *after-initial M1* $\alpha \neq$ *after-initial M1* $\beta \Longrightarrow \exists$ *io* $\in$ *set* (*get-separating-traces M1* ((α,*after-initial M1* α),(β,*after-initial M1* β)) *t*) $\cup$ (*set* (*after t* α) $\cap$ *set* (*after t* β)) . *distinguishes M1* (*after-initial M1* α) (*after-initial M1* β) *io*
and $\bigwedge \alpha\ \beta\ t$. $\alpha \in L$ *M1* $\Longrightarrow \beta \in L$ *M1* $\Longrightarrow$ *after-initial M1* $\alpha \neq$ *after-initial M1* $\beta \Longrightarrow$ *finite-tree* (*get-separating-traces M1* ((α,*after-initial M1* α),(β,*after-initial M1* β)) *t*)
shows (*L M1* = *L M2*) $\longleftrightarrow$ ((*L M1* $\cap$ *set* (*pair-framework-h-components-2 M1 m get-separating-traces c*)) = (*L M2* $\cap$ *set* (*pair-framework-h-components-2 M1 m get-separating-traces c*)))
and *finite-tree* (*pair-framework-h-components-2 M1 m get-separating-traces c*)
⟨*proof*⟩

24.4 Code Generation

```
lemma h-framework-dynamic-code[code] :
  h-framework-dynamic convergence-decisision M1 m completeInputTraces useIn-
putHeuristic = (let
      tables = (compute-ofsm-tables M1 (size M1 − 1));
    distMap = mapping-of (map (λ (q1,q2) . ((q1,q2), get-distinguishing-sequence-from-ofsm-tables-with-provid
tables M1 q1 q2))
                         (filter (λ qq . fst qq ≠ snd qq) (List.product (states-as-list M1)
(states-as-list M1))));
     distHelper = (λ q1 q2 . if q1 ∈ states M1 ∧ q2 ∈ states M1 ∧ q1 ≠ q2 then the
(Mapping.lookup distMap (q1,q2)) else get-distinguishing-sequence-from-ofsm-tables
M1 q1 q2)
    in
      h-framework  M1
                  get-state-cover-assignment
                 (handle-state-cover-dynamic completeInputTraces useInputHeuristic
distHelper)
                  sort-unverified-transitions-by-state-cover-length
               (handleUT-dynamic completeInputTraces useInputHeuristic distHelper
convergence-decisision)
                  (handle-io-pair completeInputTraces useInputHeuristic)
                  simple-cg-initial
                  simple-cg-insert
                  simple-cg-lookup-with-conv
                  simple-cg-merge
                  m)
  ⟨proof⟩

end
```

25 Implementations of the H-Method

```
theory H-Method-Implementations
imports Intermediate-Frameworks Pair-Framework ../Distinguishability Test-Suite-Representations
../OFSM-Tables-Refined HOL−Library.List-Lexorder
begin
```

25.1 Using the H-Framework

```
definition h-method-via-h-framework :: ('a::linorder,'b::linorder,'c::linorder) fsm
⇒ nat ⇒ bool ⇒ bool ⇒ ('b×'c) prefix-tree where
  h-method-via-h-framework = h-framework-dynamic (λ M V t X l . False)

definition h-method-via-h-framework-lists :: ('a::linorder,'b::linorder,'c::linorder)
fsm ⇒ nat ⇒ bool ⇒ bool ⇒ (('b×'c) × bool) list list where
  h-method-via-h-framework-lists M m completeInputTraces useInputHeuristic =
sorted-list-of-maximal-sequences-in-tree (test-suite-from-io-tree M (initial M) (h-method-via-h-framework
M m completeInputTraces useInputHeuristic))
```

lemma *h-method-via-h-framework-completeness-and-finiteness* :
fixes *M1* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm*
fixes *M2* :: (*'e*,*'b*,*'c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*h-method-via-h-framework M1 m completeInputTraces useInputHeuristic*)) = (*L M2* ∩ *set* (*h-method-via-h-framework M1 m completeInputTraces useInputHeuristic*)))
and *finite-tree* (*h-method-via-h-framework M1 m completeInputTraces useInputHeuristic*)
⟨*proof*⟩

lemma *h-method-via-h-framework-lists-completeness* :
fixes *M1* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm*
fixes *M2* :: (*'d*,*'b*,*'c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*initial M2*)) (*h-method-via-h-framework-lists M1 m completeInputTraces useInputHeuristic*)
⟨*proof*⟩

25.2 Using the Pair-Framework

25.2.1 Selection of Distinguishing Traces

fun *add-distinguishing-sequence-if-required* :: (*'a* ⇒ *'a* ⇒ (*'b* × *'c*) *list*) ⇒ (*'a*,*'b*::*linorder*,*'c*::*linorder*) *fsm* ⇒ ((*'b*×*'c*) *list* × *'a*) × ((*'b*×*'c*) *list* × *'a*) ⇒ (*'b*×*'c*) *prefix-tree* ⇒ (*'b*×*'c*) *prefix-tree* **where**
add-distinguishing-sequence-if-required dist-fun M ((α,*q1*), (β,*q2*)) *t* = (*if intersection-is-distinguishing M* (*after t* α) *q1* (*after t* β) *q2*
then empty
else insert empty (*dist-fun q1 q2*))

lemma *add-distinguishing-sequence-if-required-distinguishes* :
assumes *observable M*
and *minimal M*
and α ∈ *L M*

and $\beta \in L\ M$
and *after-initial* $M\ \alpha \neq$ *after-initial* $M\ \beta$
and $\bigwedge$ *q1 q2* . *q1* $\in$ *states* $M \Longrightarrow$ *q2* $\in$ *states* $M \Longrightarrow$ *q1* $\neq$ *q2* $\Longrightarrow$ *distinguishes M q1 q2* (*dist-fun q1 q2*)
shows $\exists$ *io* $\in$ *set* ((*add-distinguishing-sequence-if-required dist-fun M*) ((α,*after-initial M* α),(β,*after-initial M* β)) *t*) $\cup$ (*set* (*after t* α) $\cap$ *set* (*after t* β)) . *distinguishes M* (*after-initial M* α) (*after-initial M* β) *io*
⟨*proof*⟩

lemma *add-distinguishing-sequence-if-required-finite* :
finite-tree ((*add-distinguishing-sequence-if-required dist-fun M*) ((α,*after-initial M* α),(β,*after-initial M* β)) *t*)
⟨*proof*⟩

fun *add-distinguishing-sequence-and-complete-if-required* :: ('*a* ⇒ '*a* ⇒ ('*b* × '*c*) *list*) ⇒ *bool* ⇒ ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ (('*b*×'*c*) *list* × '*a*) × (('*b*×'*c*) *list* × '*a*) ⇒ ('*b*×'*c*) *prefix-tree* ⇒ ('*b*×'*c*) *prefix-tree* **where**
add-distinguishing-sequence-and-complete-if-required distFun completeInputTraces M ((α,*q1*), (β,*q2*)) *t* =
(*if intersection-is-distinguishing M* (*after t* α) *q1* (*after t* β) *q2*
then empty
else let w = *distFun q1 q2*;
T = *insert empty w*
in if completeInputTraces
then let T1 = *from-list* (*language-for-input M q1* (*map fst w*));
T2 = *from-list* (*language-for-input M q2* (*map fst w*))
in Prefix-Tree.combine T (*Prefix-Tree.combine T1 T2*)
else T)

lemma *add-distinguishing-sequence-and-complete-if-required-distinguishes* :
assumes *observable M*
and *minimal M*
and $\alpha \in L\ M$
and $\beta \in L\ M$
and *after-initial* $M\ \alpha \neq$ *after-initial* $M\ \beta$
and $\bigwedge$ *q1 q2* . *q1* $\in$ *states* $M \Longrightarrow$ *q2* $\in$ *states* $M \Longrightarrow$ *q1* $\neq$ *q2* $\Longrightarrow$ *distinguishes M q1 q2* (*dist-fun q1 q2*)
shows $\exists$ *io* $\in$ *set* ((*add-distinguishing-sequence-and-complete-if-required dist-fun c M*) ((α,*after-initial M* α),(β,*after-initial M* β)) *t*) $\cup$ (*set* (*after t* α) $\cap$ *set* (*after t* β)) . *distinguishes M* (*after-initial M* α) (*after-initial M* β) *io*
⟨*proof*⟩

lemma *add-distinguishing-sequence-and-complete-if-required-finite* :
finite-tree ((*add-distinguishing-sequence-and-complete-if-required dist-fun c M*) ((α,*after-initial M* α),(β,*after-initial M* β)) *t*)
⟨*proof*⟩

```
function find-cheapest-distinguishing-trace :: ('a,'b::linorder,'c::linorder) fsm ⇒
('a ⇒ 'a ⇒ ('b × 'c) list) ⇒ ('b×'c) list ⇒ ('b×'c) prefix-tree ⇒ 'a ⇒ ('b×'c)
prefix-tree ⇒ 'a ⇒ (('b×'c) list × nat × nat) where
  find-cheapest-distinguishing-trace M distFun ios (PT m1) q1 (PT m2) q2 =
    (let
      f = (λ (ω,l,w) (x,y) . if (x,y) ∉ list.set ios then (ω,l,w) else
           (let
             w1L = if (PT m1) = empty then 0 else 1;
             w1C = if (x,y) ∈ dom m1 then 0 else 1;
             w1 = min w1L w1C;
             w2L = if (PT m2) = empty then 0 else 1;
             w2C = if (x,y) ∈ dom m2 then 0 else 1;
             w2 = min w2L w2C;
             w' = w1 + w2
           in
             case h-obs M q1 x y of
               None ⇒ (case h-obs M q2 x y of
                 None ⇒ (ω,l,w) |
                   Some - ⇒ if w' = 0 ∨ w' ≤ w then ([(x,y)],w1C+w2C,w') else
(ω,l,w)) |
               Some q1' ⇒ (case h-obs M q2 x y of
                None ⇒ if w' = 0 ∨ w' ≤ w then ([(x,y)],w1C+w2C,w') else (ω,l,w)
|
                 Some q2' ⇒ (if q1' = q2'
                   then (ω,l,w)
                   else (case m1 (x,y) of
                     None ⇒ (case m2 (x,y) of
                       None ⇒ let ω' = distFun q1' q2';
                                  l' = 2 + 2 * length ω'
                              in if (w' < w) ∨ (w' = w ∧ l' < l) then ((x,y)#ω',l',w')
else (ω,l,w) |
                        Some t2' ⇒ let (ω'',l'',w'') = find-cheapest-distinguishing-trace
M distFun ios empty q1' t2' q2'
                                   in if (w'' + w1 < w) ∨ (w'' + w1 = w ∧ l''+1 < l)
then ((x,y)#ω'',l''+1,w''+w1) else (ω,l,w)) |
                      Some t1' ⇒ (case m2 (x,y) of
                        None ⇒ let (ω'',l'',w'') = find-cheapest-distinguishing-trace M
distFun ios t1' q1' empty q2'
                                  in if (w'' + w2 < w) ∨ (w'' + w2 = w ∧ l''+1 < l)
then ((x,y)#ω'',l''+1,w''+w2) else (ω,l,w) |
                        Some t2' ⇒ let (ω'',l'',w'') = find-cheapest-distinguishing-trace
M distFun ios t1' q1' t2' q2'
                                        in if (w'' < w) ∨ (w'' = w ∧ l'' < l) then
((x,y)#ω'',l'',w'') else (ω,l,w)))))))
     in
       foldl f (distFun q1 q2, 0, 3) ios)
  ⟨proof⟩
termination
```

⟨*proof*⟩

lemma *find-cheapest-distinguishing-trace-alt-def* :
find-cheapest-distinguishing-trace M distFun ios (PT m1) q1 (PT m2) q2 =
(let
f = (λ (ω,l,w) (x,y).
(let
w1L = if (PT m1) = empty then 0 else 1;
w1C = if (x,y) ∈ dom m1 then 0 else 1;
w1 = min w1L w1C;
w2L = if (PT m2) = empty then 0 else 1;
w2C = if (x,y) ∈ dom m2 then 0 else 1;
w2 = min w2L w2C;
w′ = w1 + w2
in
case h-obs M q1 x y of
None ⇒ (case h-obs M q2 x y of
None ⇒ (ω,l,w) |
Some - ⇒ if w′ = 0 ∨ w′ ≤ w then ([(x,y)],w1C+w2C,w′) else (ω,l,w)) |
Some q1′ ⇒ (case h-obs M q2 x y of
None ⇒ if w′ = 0 ∨ w′ ≤ w then ([(x,y)],w1C+w2C,w′) else (ω,l,w) |
Some q2′ ⇒ (if q1′ = q2′
then (ω,l,w)
else (case m1 (x,y) of
None ⇒ (case m2 (x,y) of
None ⇒ let ω′ = distFun q1′ q2′;
l′ = 2 + 2 ∗ length ω′
in if (w′ < w) ∨ (w′ = w ∧ l′ < l) then ((x,y)#ω′,l′,w′) else (ω,l,w) |
Some t2′ ⇒ let (ω′′,l′′,w′′) = find-cheapest-distinguishing-trace M distFun ios empty q1′ t2′ q2′
in if (w′′ + w1 < w) ∨ (w′′ + w1 = w ∧ l′′+1 < l) then ((x,y)#ω′′,l′′+1,w′′+w1) else (ω,l,w)) |
Some t1′ ⇒ (case m2 (x,y) of
None ⇒ let (ω′′,l′′,w′′) = find-cheapest-distinguishing-trace M distFun ios t1′ q1′ empty q2′
in if (w′′ + w2 < w) ∨ (w′′ + w2 = w ∧ l′′+1 < l) then ((x,y)#ω′′,l′′+1,w′′+w2) else (ω,l,w) |
Some t2′ ⇒ let (ω′′,l′′,w′′) = find-cheapest-distinguishing-trace M distFun ios t1′ q1′ t2′ q2′
in if (w′′ < w) ∨ (w′′ = w ∧ l′′ < l) then ((x,y)#ω′′,l′′,w′′) else (ω,l,w)))))))
in

foldl f (distFun q1 q2, 0, 3) ios)
(**is** *find-cheapest-distinguishing-trace M distFun ios (PT m1) q1 (PT m2) q2 = ?find-cheapest-distinguishing-trace*)

⟨*proof*⟩

lemma *find-cheapest-distinguishing-trace-code*[*code*] :
find-cheapest-distinguishing-trace M distFun ios (MPT m1) q1 (MPT m2) q2 =
(let
f = (λ (ω,l,w) (x,y) .
(let
w1L = if is-leaf (MPT m1) then 0 else 1;
w1C = if (x,y) ∈ Mapping.keys m1 then 0 else 1;
w1 = min w1L w1C;
w2L = if is-leaf (MPT m2) then 0 else 1;
w2C = if(x,y) ∈ Mapping.keys m2 then 0 else 1;
w2 = min w2L w2C;
w′ = w1 + w2
in
case h-obs M q1 x y of
None ⇒ (case h-obs M q2 x y of
None ⇒ (ω,l,w) |
Some - ⇒ if w′ = 0 ∨ w′ ≤ w then ([(x,y)],w1C+w2C,w′) else (ω,l,w)) |
Some q1′ ⇒ (case h-obs M q2 x y of
None ⇒ if w′ = 0 ∨ w′ ≤ w then ([(x,y)],w1C+w2C,w′) else (ω,l,w) |
Some q2′ ⇒ (if q1′ = q2′
then (ω,l,w)
else (case Mapping.lookup m1 (x,y) of
None ⇒ (case Mapping.lookup m2 (x,y) of
None ⇒ let ω′ = distFun q1′ q2′;
l′ = 2 + 2 ∗ length ω′
in if (w′ < w) ∨ (w′ = w ∧ l′ < l) then ((x,y)#ω′,l′,w′) else (ω,l,w) |
Some t2′ ⇒ let (ω′′,l′′,w′′) = find-cheapest-distinguishing-trace M distFun ios empty q1′ t2′ q2′
in if (w′′ + w1 < w) ∨ (w′′ + w1 = w ∧ l′′+1 < l) then ((x,y)#ω′′,l′′+1,w′′+w1) else (ω,l,w)) |
Some t1′ ⇒ (case Mapping.lookup m2 (x,y) of
None ⇒ let (ω′′,l′′,w′′) = find-cheapest-distinguishing-trace M distFun ios t1′ q1′ empty q2′
in if (w′′ + w2 < w) ∨ (w′′ + w2 = w ∧ l′′+1 < l) then ((x,y)#ω′′,l′′+1,w′′+w2) else (ω,l,w) |
Some t2′ ⇒ let (ω′′,l′′,w′′) = find-cheapest-distinguishing-trace M distFun ios t1′ q1′ t2′ q2′
in if (w′′ < w) ∨ (w′′ = w ∧ l′′ < l) then ((x,y)#ω′′,l′′,w′′) else (ω,l,w))))))

in
foldl f (*distFun q1 q2*, *0*, *3*) *ios*)
⟨*proof*⟩

lemma *find-cheapest-distinguishing-trace-is-distinguishing-trace* :
assumes *observable M*
and *minimal M*
and $q1 \in$ *states M*
and $q2 \in$ *states M*
and $q1 \neq q2$
and $\bigwedge$ *q1 q2* . $q1 \in$ *states M* $\Longrightarrow$ $q2 \in$ *states M* $\Longrightarrow$ $q1 \neq q2$ $\Longrightarrow$ *distinguishes M q1 q2* (*distFun q1 q2*)
shows *distinguishes M q1 q2* (*fst* (*find-cheapest-distinguishing-trace M distFun ios t1 q1 t2 q2*))
⟨*proof*⟩

fun *add-cheapest-distinguishing-trace* :: ($'a \Rightarrow 'a \Rightarrow ('b \times 'c)$ *list*) $\Rightarrow$ *bool* $\Rightarrow$ ($'a$::*linorder*,$'b$::*linorder*,$'c$::*linorder*) *fsm* $\Rightarrow$ (($'b \times 'c$) *list* $\times$ $'a$) $\times$ (($'b \times 'c$) *list* $\times$ $'a$) $\Rightarrow$ ($'b \times 'c$) *prefix-tree* $\Rightarrow$ ($'b \times 'c$) *prefix-tree* **where**
add-cheapest-distinguishing-trace distFun completeInputTraces M ((α,*q1*), (β,*q2*)) *t* =
(*let w* = (*fst* (*find-cheapest-distinguishing-trace M distFun* (*List.product* (*inputs-as-list M*) (*outputs-as-list M*)) (*after t* α) *q1* (*after t* β) *q2*));
T = *insert empty w*
in if completeInputTraces
then let T1 = *complete-inputs-to-tree M q1* (*outputs-as-list M*) (*map fst w*);
T2 = *complete-inputs-to-tree M q2* (*outputs-as-list M*) (*map fst w*)
in Prefix-Tree.combine T (*Prefix-Tree.combine T1 T2*)
else T)

lemma *add-cheapest-distinguishing-trace-distinguishes* :
assumes *observable M*
and *minimal M*
and $\alpha \in L\ M$
and $\beta \in L\ M$
and *after-initial M* $\alpha \neq$ *after-initial M* β
and $\bigwedge$ *q1 q2* . $q1 \in$ *states M* $\Longrightarrow$ $q2 \in$ *states M* $\Longrightarrow$ $q1 \neq q2$ $\Longrightarrow$ *distinguishes M q1 q2* (*dist-fun q1 q2*)
shows $\exists$ *io* $\in$ *set* ((*add-cheapest-distinguishing-trace dist-fun c M*) ((α,*after-initial M* α),(β,*after-initial M* β)) *t*) $\cup$ (*set* (*after t* α) $\cap$ *set* (*after t* β)) . *distinguishes M* (*after-initial M* α) (*after-initial M* β) *io*
⟨*proof*⟩

lemma *add-cheapest-distinguishing-trace-finite* :
finite-tree ((*add-cheapest-distinguishing-trace dist-fun c M*) ((α,*after-initial M*

α),(*β*,*after-initial M β*)) *t*)
⟨*proof*⟩

25.2.2 Implementation

definition *h-method-via-pair-framework* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* ⇒ *nat* ⇒ (*'b*×*'c*) *prefix-tree* **where**
h-method-via-pair-framework M m = *pair-framework-h-components M m* (*add-distinguishing-sequence-if-requi*
(*get-distinguishing-sequence-from-ofsm-tables M*))

lemma *h-method-via-pair-framework-completeness-and-finiteness* :
assumes *observable M*
and *observable I*
and *minimal M*
and *size I* ≤ *m*
and *m* ≥ *size-r M*
and *inputs I* = *inputs M*
and *outputs I* = *outputs M*
shows (*L M* = *L I*) ⟷ (*L M* ∩ *set* (*h-method-via-pair-framework M m*) = *L I* ∩ *set* (*h-method-via-pair-framework M m*))
and *finite-tree* (*h-method-via-pair-framework M m*)
⟨*proof*⟩

definition *h-method-via-pair-framework-2* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* ⇒ *nat* ⇒ *bool* ⇒ (*'b*×*'c*) *prefix-tree* **where**
h-method-via-pair-framework-2 M m c = *pair-framework-h-components M m* (*add-distinguishing-sequence-and*
(*get-distinguishing-sequence-from-ofsm-tables M*) *c*)

lemma *h-method-via-pair-framework-2-completeness-and-finiteness* :
assumes *observable M*
and *observable I*
and *minimal M*
and *size I* ≤ *m*
and *m* ≥ *size-r M*
and *inputs I* = *inputs M*
and *outputs I* = *outputs M*
shows (*L M* = *L I*) ⟷ (*L M* ∩ *set* (*h-method-via-pair-framework-2 M m c*) = *L I* ∩ *set* (*h-method-via-pair-framework-2 M m c*))
and *finite-tree* (*h-method-via-pair-framework-2 M m c*)
⟨*proof*⟩

definition *h-method-via-pair-framework-3* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* ⇒ *nat* ⇒ *bool* ⇒ *bool* ⇒ (*'b*×*'c*) *prefix-tree* **where**
h-method-via-pair-framework-3 M m c1 c2 = *pair-framework-h-components-2 M m* (*add-cheapest-distinguishing-trace* (*get-distinguishing-sequence-from-ofsm-tables M*) *c2*) *c1*

lemma *h-method-via-pair-framework-3-completeness-and-finiteness* :
assumes *observable M*

and *observable I*
and *minimal M*
and *size I* ≤ *m*
and *m* ≥ *size-r M*
and *inputs I* = *inputs M*
and *outputs I* = *outputs M*
shows (*L M* = *L I*) ⟷ (*L M* ∩ *set* (*h-method-via-pair-framework-3 M m c1 c2*) = *L I* ∩ *set* (*h-method-via-pair-framework-3 M m c1 c2*))
and *finite-tree* (*h-method-via-pair-framework-3 M m c1 c2*)
⟨*proof*⟩

definition *h-method-via-pair-framework-lists* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ *nat* ⇒ (('*b*×'*c*) × *bool*) *list list* **where**
h-method-via-pair-framework-lists M m = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*h-method-via-pair-framework M m*))

lemma *h-method-implementation-lists-completeness* :
assumes *observable M*
and *observable I*
and *minimal M*
and *size I* ≤ *m*
and *m* ≥ *size-r M*
and *inputs I* = *inputs M*
and *outputs I* = *outputs M*
shows (*L M* = *L I*) ⟷ *list-all* (*passes-test-case I* (*initial I*)) (*h-method-via-pair-framework-lists M m*)
⟨*proof*⟩

25.2.3 Code Equations

lemma *h-method-via-pair-framework-code*[*code*] :
h-method-via-pair-framework M m = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
distMap = *mapping-of* (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provided-tables M q1 q2*))
(*filter* (λ *qq* . *fst qq* ≠ *snd qq*) (*List.product* (*states-as-list M*) (*states-as-list M*))));
distHelper = (λ *q1 q2* . *if q1* ∈ *states M* ∧ *q2* ∈ *states M* ∧ *q1* ≠ *q2 then the* (*Mapping.lookup distMap* (*q1*,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables M q1 q2*);
distFun = *add-distinguishing-sequence-if-required distHelper*
in pair-framework-h-components M m distFun)
⟨*proof*⟩

lemma *h-method-via-pair-framework-2-code*[*code*] :
h-method-via-pair-framework-2 M m c = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
distMap = *mapping-of* (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provide*

tables M q1 q2))
(*filter* (λ *qq* . *fst qq* ≠ *snd qq*) (*List.product* (*states-as-list M*)
(*states-as-list M*))));
distHelper = (λ *q1 q2* . *if q1* ∈ *states M* ∧ *q2* ∈ *states M* ∧ *q1* ≠ *q2 then the*
(*Mapping.lookup distMap* (*q1*,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables*
M q1 q2);
distFun = *add-distinguishing-sequence-and-complete-if-required distHelper c*
in pair-framework-h-components M m distFun)
⟨*proof*⟩

lemma *h-method-via-pair-framework-3-code*[*code*] :
h-method-via-pair-framework-3 M m c1 c2 = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
distMap = *mapping-of* (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provide*
tables M q1 q2))
(*filter* (λ *qq* . *fst qq* ≠ *snd qq*) (*List.product* (*states-as-list M*)
(*states-as-list M*))));
distHelper = (λ *q1 q2* . *if q1* ∈ *states M* ∧ *q2* ∈ *states M* ∧ *q1* ≠ *q2 then the*
(*Mapping.lookup distMap* (*q1*,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables*
M q1 q2);
distFun = *add-cheapest-distinguishing-trace distHelper c2*
in pair-framework-h-components-2 M m distFun c1)
⟨*proof*⟩

end

26 Implementations of the HSI-Method

theory *HSI-Method-Implementations*
imports *Intermediate-Frameworks Pair-Framework ../Distinguishability Test-Suite-Representations*
../OFSM-Tables-Refined HOL−Library.List-Lexorder
begin

26.1 Using the H-Framework

definition *hsi-method-via-h-framework* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
⇒ *nat* ⇒ ('*b*×'*c*) *prefix-tree* **where**
hsi-method-via-h-framework M m = *h-framework-static-with-empty-graph M* (λ *k*
q . *get-HSI M q*) *m*

definition *hsi-method-via-h-framework-lists* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*)
fsm ⇒ *nat* ⇒ (('*b*×'*c*) × *bool*) *list list* **where**
hsi-method-via-h-framework-lists M m = *sorted-list-of-maximal-sequences-in-tree*
(*test-suite-from-io-tree M* (*initial M*) (*hsi-method-via-h-framework M m*))

lemma *hsi-method-via-h-framework-completeness-and-finiteness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*e*,'*b*,'*c*) *fsm*
assumes *observable M1*

and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*hsi-method-via-h-framework M1 m*)) = (*L M2* ∩ *set* (*hsi-method-via-h-framework M1 m*)))
and *finite-tree* (*hsi-method-via-h-framework M1 m*)
⟨*proof*⟩

lemma *hsi-method-via-h-framework-lists-completeness* :
fixes *M1* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm*
fixes *M2* :: (*'d*,*'b*,*'c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*initial M2*)) (*hsi-method-via-h-framework-lists M1 m*)
⟨*proof*⟩

26.2 Using the SPY-Framework

definition *hsi-method-via-spy-framework* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* ⇒ *nat* ⇒ (*'b*×*'c*) *prefix-tree* **where**
hsi-method-via-spy-framework M m = *spy-framework-static-with-empty-graph M* (λ *k q* . *get-HSI M q*) *m*

lemma *hsi-method-via-spy-framework-completeness-and-finiteness* :
fixes *M1* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm*
fixes *M2* :: (*'d*,*'b*,*'c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*hsi-method-via-spy-framework M1 m*)) = (*L M2* ∩ *set* (*hsi-method-via-spy-framework M1 m*)))
and *finite-tree* (*hsi-method-via-spy-framework M1 m*)
⟨*proof*⟩

definition *hsi-method-via-spy-framework-lists* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* ⇒ *nat* ⇒ ((*'b*×*'c*) × *bool*) *list list* **where**
hsi-method-via-spy-framework-lists M m = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*hsi-method-via-spy-framework M m*))

lemma *hsi-method-via-spy-framework-lists-completeness* :
fixes *M1* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm*
fixes *M2* :: (*'d*,*'b*,*'c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*initial M2*)) (*hsi-method-via-spy-framework-lists M1 m*)
⟨*proof*⟩

26.3 Using the Pair-Framework

definition *hsi-method-via-pair-framework* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* ⇒ *nat* ⇒ (*'b*×*'c*) *prefix-tree* **where**
hsi-method-via-pair-framework M m = *pair-framework-h-components M m* (*add-distinguishing-sequence*)

lemma *hsi-method-via-pair-framework-completeness-and-finiteness* :
assumes *observable M*
and *observable I*
and *minimal M*
and *size I* ≤ *m*
and *m* ≥ *size-r M*
and *inputs I* = *inputs M*
and *outputs I* = *outputs M*
shows (*L M* = *L I*) ⟷ (*L M* ∩ *set* (*hsi-method-via-pair-framework M m*) = *L I* ∩ *set* (*hsi-method-via-pair-framework M m*))
and *finite-tree* (*hsi-method-via-pair-framework M m*)
⟨*proof*⟩

definition *hsi-method-via-pair-framework-lists* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* ⇒ *nat* ⇒ ((*'b*×*'c*) × *bool*) *list list* **where**
hsi-method-via-pair-framework-lists M m = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*hsi-method-via-pair-framework M m*))

lemma *hsi-method-implementation-lists-completeness* :
assumes *observable M*
and *observable I*

and *minimal M*
and *size I* ≤ *m*
and *m* ≥ *size-r M*
and *inputs I* = *inputs M*
and *outputs I* = *outputs M*
shows (*L M* = *L I*) ⟷ *list-all* (*passes-test-case I* (*initial I*)) (*hsi-method-via-pair-framework-lists M m*)
⟨*proof*⟩

26.4 Code Generation

lemma *hsi-method-via-pair-framework-code*[*code*] :
hsi-method-via-pair-framework M m = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
distMap = *mapping-of* (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provid*
tables M q1 q2))
(*filter* (λ *qq* . *fst qq* ≠ *snd qq*) (*List.product* (*states-as-list M*)
(*states-as-list M*))));
distHelper = (λ *q1 q2* . *if q1* ∈ *states M* ∧ *q2* ∈ *states M* ∧ *q1* ≠ *q2 then the*
(*Mapping.lookup distMap* (*q1*,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables*
M q1 q2);
distFun = (λ *M* ((*io1*,*q1*),(*io2*,*q2*)) *t* . *insert empty* (*distHelper q1 q2*))
in pair-framework-h-components M m distFun)
⟨*proof*⟩

lemma *hsi-method-via-spy-framework-code*[*code*] :
hsi-method-via-spy-framework M m = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
distMap = *mapping-of* (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provide*
tables M q1 q2))
(*filter* (λ *qq* . *fst qq* ≠ *snd qq*) (*List.product* (*states-as-list M*)
(*states-as-list M*))));
distHelper = (λ *q1 q2* . *if q1* ∈ *states M* ∧ *q2* ∈ *states M* ∧ *q1* ≠ *q2 then the*
(*Mapping.lookup distMap* (*q1*,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables*
M q1 q2);

hsiMap = *mapping-of* (*map* (λ *q* . (*q*,*from-list* (*map* (λ*q'* . *distHelper q q'*) (*filter*
((≠) *q*) (*states-as-list M*))))) (*states-as-list M*));
distFun = (λ *k q* . *if q* ∈ *states M then the* (*Mapping.lookup hsiMap q*) *else*
get-HSI M q)
in spy-framework-static-with-empty-graph M distFun m)
(**is** *?f1* = *?f2*)
⟨*proof*⟩

lemma *hsi-method-via-h-framework-code*[*code*] :
hsi-method-via-h-framework M m = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
distMap = *mapping-of* (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provid*
tables M q1 q2))

(*filter* (λ *qq* . *fst qq* ≠ *snd qq*) (*List.product* (*states-as-list M*) (*states-as-list M*))));
distHelper = (λ *q1 q2* . *if q1* ∈ *states M* ∧ *q2* ∈ *states M* ∧ *q1* ≠ *q2 then the* (*Mapping.lookup distMap* (*q1*,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables M q1 q2*);

hsiMap = *mapping-of* (*map* (λ *q* . (*q*,*from-list* (*map* (λ*q'* . *distHelper q q'*) (*filter* ((≠) *q*) (*states-as-list M*))))) (*states-as-list M*));
distFun = (λ *k q* . *if q* ∈ *states M then the* (*Mapping.lookup hsiMap q*) *else get-HSI M q*)
in h-framework-static-with-empty-graph M distFun m)
(**is** *?f1* = *?f2*)
⟨*proof*⟩

end

27 Implementations of the Partial-S-Method

theory *Partial-S-Method-Implementations*
imports *Intermediate-Frameworks*
begin

27.1 Using the H-Framework

fun *distance-at-most* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* ⇒ *'a* ⇒ *'a* ⇒ *nat* ⇒ *bool* **where**
distance-at-most M q1 q2 0 = (*q1* = *q2*) |
distance-at-most M q1 q2 (*Suc k*) = ((*q1* = *q2*) ∨ (∃ *x* ∈ *inputs M* . ∃ (*y*,*q1'*) ∈ *h M* (*q1*,*x*) . *distance-at-most M q1' q2 k*))

definition *do-establish-convergence* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* ⇒ (*'a*,*'b*,*'c*) *state-cover-assignment* ⇒ (*'a*,*'b*,*'c*) *transition* ⇒ (*'a*,*'b*,*'c*) *transition list* ⇒ *nat* ⇒ *bool* **where**
do-establish-convergence M V t X l = (*find* (λ *t'* . *distance-at-most M* (*t-target t*) (*t-source t'*) *l*) *X* ≠ *None*)

definition *partial-s-method-via-h-framework* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* ⇒ *nat* ⇒ *bool* ⇒ *bool* ⇒ (*'b*×*'c*) *prefix-tree* **where**
partial-s-method-via-h-framework = *h-framework-dynamic do-establish-convergence*

definition *partial-s-method-via-h-framework-lists* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* ⇒ *nat* ⇒ *bool* ⇒ *bool* ⇒ ((*'b*×*'c*) × *bool*) *list list* **where**
partial-s-method-via-h-framework-lists M m completeInputTraces useInputHeuristic = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*partial-s-method-via-h-framework M m completeInputTraces useInputHeuristic*))

lemma *partial-s-method-via-h-framework-completeness-and-finiteness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*e*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*partial-s-method-via-h-framework M1 m completeInputTraces useInputHeuristic*)) = (*L M2* ∩ *set* (*partial-s-method-via-h-framework M1 m completeInputTraces useInputHeuristic*)))
and *finite-tree* (*partial-s-method-via-h-framework M1 m completeInputTraces useInputHeuristic*)
⟨*proof*⟩

lemma *partial-s-method-via-h-framework-lists-completeness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*d*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*initial M2*)) (*partial-s-method-via-h-framework-lists M1 m completeInputTraces useInputHeuristic*)
⟨*proof*⟩

end

28 Implementations of the SPY-Method

theory *SPY-Method-Implementations*
imports *Intermediate-Frameworks Pair-Framework ../Distinguishability Test-Suite-Representations ../OFSM-Tables-Refined HOL−Library.List-Lexorder*
begin

28.1 Using the H-Framework

definition *spy-method-via-h-framework* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ *nat* ⇒ ('*b*×'*c*) *prefix-tree* **where**
spy-method-via-h-framework M m = *h-framework-static-with-simple-graph M* (λ *k q* . *get-HSI M q*) *m*

definition *spy-method-via-h-framework-lists* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ *nat* ⇒ (('*b*×'*c*) × *bool*) *list list* **where**
spy-method-via-h-framework-lists M m = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*spy-method-via-h-framework M m*))

lemma *spy-method-via-h-framework-completeness-and-finiteness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*e*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*spy-method-via-h-framework M1 m*)) = (*L M2* ∩ *set* (*spy-method-via-h-framework M1 m*)))
and *finite-tree* (*spy-method-via-h-framework M1 m*)
⟨*proof*⟩

lemma *spy-method-via-h-framework-lists-completeness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*d*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*initial M2*)) (*spy-method-via-h-framework-lists M1 m*)
⟨*proof*⟩

28.2 Using the SPY-Framework

definition *spy-method-via-spy-framework* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ *nat* ⇒ ('*b*×'*c*) *prefix-tree* **where**
spy-method-via-spy-framework M m = *spy-framework-static-with-simple-graph M* (λ *k q* . *get-HSI M q*) *m*

lemma *spy-method-via-spy-framework-completeness-and-finiteness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*d*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*

and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*spy-method-via-spy-framework M1 m*)) = (*L M2* ∩ *set* (*spy-method-via-spy-framework M1 m*)))
and *finite-tree* (*spy-method-via-spy-framework M1 m*)
⟨*proof*⟩

definition *spy-method-via-spy-framework-lists* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ *nat* ⇒ (('*b*×'*c*) × *bool*) *list list* **where**
spy-method-via-spy-framework-lists M m = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*spy-method-via-spy-framework M m*))

lemma *spy-method-via-spy-framework-lists-completeness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*d*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*initial M2*)) (*spy-method-via-spy-framework-lists M1 m*)
⟨*proof*⟩

28.3 Code Generation

lemma *spy-method-via-spy-framework-code*[*code*] :
spy-method-via-spy-framework M m = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
distMap = *mapping-of* (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provided-tables M q1 q2*))
(*filter* (λ *qq* . *fst qq* ≠ *snd qq*) (*List.product* (*states-as-list M*) (*states-as-list M*))));
distHelper = (λ *q1 q2* . *if q1* ∈ *states M* ∧ *q2* ∈ *states M* ∧ *q1* ≠ *q2 then the* (*Mapping.lookup distMap* (*q1*,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables M q1 q2*);

hsiMap = *mapping-of* (*map* (λ *q* . (*q*,*from-list* (*map* (λ*q*' . *distHelper q q*') (*filter* ((≠) *q*) (*states-as-list M*))))) (*states-as-list M*));
distFun = (λ *k q* . *if q* ∈ *states M then the* (*Mapping.lookup hsiMap q*) *else get-HSI M q*)
in spy-framework-static-with-simple-graph M distFun m)

(**is** *?f1 = ?f2*)
⟨*proof*⟩

lemma *spy-method-via-h-framework-code*[*code*] :
spy-method-via-h-framework M m = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
distMap = *mapping-of* (*map* (λ (*q1* ,*q2*) . ((*q1* ,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provid*
tables M q1 q2))
(*filter* (λ *qq* . *fst qq* ≠ *snd qq*) (*List.product* (*states-as-list M*)
(*states-as-list M*))));
distHelper = (λ *q1 q2* . *if q1* ∈ *states M* ∧ *q2* ∈ *states M* ∧ *q1* ≠ *q2 then the*
(*Mapping.lookup distMap* (*q1* ,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables*
M q1 q2);

hsiMap = *mapping-of* (*map* (λ *q* . (*q*,*from-list* (*map* (λ*q'* . *distHelper q q'*)
(*filter* ((≠) *q*) (*states-as-list M*))))) (*states-as-list M*));
distFun = (λ *k q* . *if q* ∈ *states M then the* (*Mapping.lookup hsiMap q*) *else*
get-HSI M q)
in h-framework-static-with-simple-graph M distFun m)
(**is** *?f1 = ?f2*)
⟨*proof*⟩

end

29 Implementations of the SPYH-Method

theory *SPYH-Method-Implementations*
imports *Intermediate-Frameworks*
begin

29.1 Using the H-Framework

definition *spyh-method-via-h-framework* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm*
⇒ *nat* ⇒ *bool* ⇒ *bool* ⇒ (*'b*×*'c*) *prefix-tree* **where**
spyh-method-via-h-framework = *h-framework-dynamic* (λ *M V t X l* . *True*)

definition *spyh-method-via-h-framework-lists* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*)
fsm ⇒ *nat* ⇒ *bool* ⇒ *bool* ⇒ ((*'b*×*'c*) × *bool*) *list list* **where**
spyh-method-via-h-framework-lists M m completeInputTraces useInputHeuristic =
sorted-list-of-maximal-sequences-in-tree (*test-suite-from-io-tree M* (*initial M*) (*spyh-method-via-h-framework*
M m completeInputTraces useInputHeuristic))

lemma *spyh-method-via-h-framework-completeness-and-finiteness* :
fixes *M1* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm*
fixes *M2* :: (*'e*,*'b*,*'c*) *fsm*
assumes *observable M1*
and *observable M2*

and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*spyh-method-via-h-framework M1 m completeInputTraces useInputHeuristic*)) = (*L M2* ∩ *set* (*spyh-method-via-h-framework M1 m completeInputTraces useInputHeuristic*)))
and *finite-tree* (*spyh-method-via-h-framework M1 m completeInputTraces useInputHeuristic*)
⟨*proof*⟩

lemma *spyh-method-via-h-framework-lists-completeness* :
fixes *M1* :: (*'a::linorder,'b::linorder,'c::linorder*) *fsm*
fixes *M2* :: (*'d,'b,'c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*initial M2*)) (*spyh-method-via-h-framework-lists M1 m completeInputTraces useInputHeuristic*)
⟨*proof*⟩

29.2 Using the SPY-Framework

definition *spyh-method-via-spy-framework* :: (*'a::linorder,'b::linorder,'c::linorder*) *fsm* ⇒ *nat* ⇒ *bool* ⇒ *bool* ⇒ (*'b×'c*) *prefix-tree* **where**
spyh-method-via-spy-framework M1 m completeInputTraces useInputHeuristic =
spy-framework M1
get-state-cover-assignment
(*handle-state-cover-dynamic completeInputTraces useInputHeuristic* (*get-distinguishing-sequence-from-ofsm-tables M1*))
sort-unverified-transitions-by-state-cover-length
(*establish-convergence-dynamic completeInputTraces useInputHeuristic* (*get-distinguishing-sequence-from-ofsm-tables M1*))
(*handle-io-pair completeInputTraces useInputHeuristic*)
simple-cg-initial
simple-cg-insert
simple-cg-lookup-with-conv
simple-cg-merge
m

lemma *spyh-method-via-spy-framework-completeness-and-finiteness* :
fixes *M1* :: (*'a::linorder,'b::linorder,'c::linorder*) *fsm*

fixes *M2* :: (*'d*,*'b*,*'c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*spyh-method-via-spy-framework M1 m completeInputTraces useInputHeuristic*)) = (*L M2* ∩ *set* (*spyh-method-via-spy-framework M1 m completeInputTraces useInputHeuristic*)))
and *finite-tree* (*spyh-method-via-spy-framework M1 m completeInputTraces useInputHeuristic*)
⟨*proof*⟩

definition *spyh-method-via-spy-framework-lists* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* ⇒ *nat* ⇒ *bool* ⇒ *bool* ⇒ ((*'b*×*'c*) × *bool*) *list list* **where**
spyh-method-via-spy-framework-lists M m completeInputTraces useInputHeuristic = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*spyh-method-via-spy-framework M m completeInputTraces useInputHeuristic*))

lemma *spyh-method-via-spy-framework-lists-completeness* :
fixes *M1* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm*
fixes *M2* :: (*'d*,*'b*,*'c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*initial M2*)) (*spyh-method-via-spy-framework-lists M1 m completeInputTraces useInputHeuristic*)
⟨*proof*⟩

29.3 Code Generation

lemma *spyh-method-via-spy-framework-code*[*code*] :
spyh-method-via-spy-framework M1 m completeInputTraces useInputHeuristic =
(*let*
tables = (*compute-ofsm-tables M1* (*size M1* − *1*));
distMap = *mapping-of* (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provid*
tables M1 q1 q2))
(*filter* (λ *qq* . *fst qq* ≠ *snd qq*) (*List.product* (*states-as-list M1*)
(*states-as-list M1*))));

distHelper = (λ *q1 q2* . *if q1* ∈ *states M1* ∧ *q2* ∈ *states M1* ∧ *q1* ≠ *q2 then the* (*Mapping.lookup distMap* (*q1*,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables M1 q1 q2*)
in
spy-framework M1
get-state-cover-assignment
(*handle-state-cover-dynamic completeInputTraces useInputHeuristic distHelper*)
sort-unverified-transitions-by-state-cover-length
(*establish-convergence-dynamic completeInputTraces useInputHeuristic distHelper*)
(*handle-io-pair completeInputTraces useInputHeuristic*)
simple-cg-initial
simple-cg-insert
simple-cg-lookup-with-conv
simple-cg-merge
m)
⟨*proof*⟩

end

30 Refined Code Generation for Test Suites

This theory provides alternative code equations for selected functions on test suites. Currently only Mapping via RBT is supported.

theory *Test-Suite-Representations-Refined*
imports *Test-Suite-Representations ../Prefix-Tree-Refined ../Util-Refined*
begin

declare [[*code drop*: *Test-Suite-Representations.test-suite-from-io-tree*]]

lemma *test-suite-from-io-tree-refined*[*code*] :
fixes *M* :: ('*a*,'*b* :: *ccompare*, '*c* :: *ccompare*) *fsm*
and *m* :: (('*b*×'*c*), ('*b*×'*c*) *prefix-tree*) *mapping-rbt*
shows *test-suite-from-io-tree M q* (*MPT* (*RBT-Mapping m*))
= (*case ID CCOMPARE*(('*b* × '*c*)) *of*
None ⇒ *Code.abort* (*STR* ''*test-suite-from-io-tree RBT-set*: *ccompare* = *None*'') (λ- . *test-suite-from-io-tree M q* (*MPT* (*RBT-Mapping m*))) |
Some - ⇒ *MPT* (*Mapping.tabulate* (*map* (λ((*x*,*y*),*t*) . ((*x*,*y*),*h-obs M q x y* ≠ *None*)) (*RBT-Mapping2.entries m*)) (λ ((*x*,*y*),*b*) . *case h-obs M q x y of None* ⇒ *Prefix-Tree.empty* | *Some q*' ⇒ *test-suite-from-io-tree M q*' (*case RBT-Mapping2.lookup m* (*x*,*y*) *of Some t*' ⇒ *t*'))))
⟨*proof*⟩

end

31 Implementations of the W-Method

theory *W-Method-Implementations*
imports *Intermediate-Frameworks Pair-Framework ../Distinguishability Test-Suite-Representations ../OFSM-Tables-Refined HOL−Library.List-Lexorder*
begin

31.1 Using the H-Framework

definition *w-method-via-h-framework* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ *nat* ⇒ ('*b*×'*c*) *prefix-tree* **where**
w-method-via-h-framework M m = *h-framework-static-with-empty-graph M* (λ *k q* . *distinguishing-set M*) *m*

definition *w-method-via-h-framework-lists* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ *nat* ⇒ (('*b*×'*c*) × *bool*) *list list* **where**
w-method-via-h-framework-lists M m = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*w-method-via-h-framework M m*))

lemma *w-method-via-h-framework-completeness-and-finiteness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*e*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*w-method-via-h-framework M1 m*)) = (*L M2* ∩ *set* (*w-method-via-h-framework M1 m*)))
and *finite-tree* (*w-method-via-h-framework M1 m*)
⟨*proof*⟩

lemma *w-method-via-h-framework-lists-completeness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*d*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*initial M2*)) (*w-method-via-h-framework-lists M1 m*)
⟨*proof*⟩

definition *w-method-via-h-framework-2* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ *nat* ⇒ ('*b*×'*c*) *prefix-tree* **where**
w-method-via-h-framework-2 M m = *h-framework-static-with-empty-graph M* (λ *k q* . *distinguishing-set-reduced M*) *m*

definition *w-method-via-h-framework-2-lists* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ *nat* ⇒ (('*b*×'*c*) × *bool*) *list list* **where**
w-method-via-h-framework-2-lists M m = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*w-method-via-h-framework-2 M m*))

lemma *w-method-via-h-framework-2-completeness-and-finiteness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*e*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*w-method-via-h-framework-2 M1 m*)) = (*L M2* ∩ *set* (*w-method-via-h-framework-2 M1 m*)))
and *finite-tree* (*w-method-via-h-framework-2 M1 m*)
⟨*proof*⟩

lemma *w-method-via-h-framework-lists-2-completeness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*d*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*initial M2*)) (*w-method-via-h-framework-2-lists M1 m*)
⟨*proof*⟩

31.2 Using the SPY-Framework

definition *w-method-via-spy-framework* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ *nat* ⇒ ('*b*×'*c*) *prefix-tree* **where**
w-method-via-spy-framework M m = *spy-framework-static-with-empty-graph M* (λ *k q* . *distinguishing-set M*) *m*

lemma *w-method-via-spy-framework-completeness-and-finiteness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*d*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*w-method-via-spy-framework M1 m*)) = (*L M2* ∩ *set* (*w-method-via-spy-framework M1 m*)))
and *finite-tree* (*w-method-via-spy-framework M1 m*)
⟨*proof*⟩

definition *w-method-via-spy-framework-lists* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ *nat* ⇒ (('*b*×'*c*) × *bool*) *list list* **where**
w-method-via-spy-framework-lists M m = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*w-method-via-spy-framework M m*))

lemma *w-method-via-spy-framework-lists-completeness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*d*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*initial M2*)) (*w-method-via-spy-framework-lists M1 m*)
⟨*proof*⟩

31.3 Using the Pair-Framework

definition *w-method-via-pair-framework* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ *nat* ⇒ ('*b*×'*c*) *prefix-tree* **where**
w-method-via-pair-framework M m = *pair-framework-h-components M m add-distinguishing-set*

lemma *w-method-via-pair-framework-completeness-and-finiteness* :
assumes *observable M*
and *observable I*
and *minimal M*
and *size I* ≤ *m*

and *m* ≥ *size-r M*
and *inputs I* = *inputs M*
and *outputs I* = *outputs M*
shows (*L M* = *L I*) ⟷ (*L M* ∩ *set* (*w-method-via-pair-framework M m*) = *L I* ∩ *set* (*w-method-via-pair-framework M m*))
and *finite-tree* (*w-method-via-pair-framework M m*)
⟨*proof*⟩

definition *w-method-via-pair-framework-lists* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ *nat* ⇒ (('*b*×'*c*) × *bool*) *list list* **where**
w-method-via-pair-framework-lists M m = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*w-method-via-pair-framework M m*))

lemma *w-method-implementation-lists-completeness* :
assumes *observable M*
and *observable I*
and *minimal M*
and *size I* ≤ *m*
and *m* ≥ *size-r M*
and *inputs I* = *inputs M*
and *outputs I* = *outputs M*
shows (*L M* = *L I*) ⟷ *list-all* (*passes-test-case I* (*initial I*)) (*w-method-via-pair-framework-lists M m*)
⟨*proof*⟩

31.4 Code Generation

lemma *w-method-via-pair-framework-code*[*code*] :
w-method-via-pair-framework M m = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
distMap = *mapping-of* (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provid*
tables M q1 q2))
(*filter* (λ *qq* . *fst qq* ≠ *snd qq*) (*List.product* (*states-as-list M*) (*states-as-list M*))));
distHelper = (λ *q1 q2* . *if q1* ∈ *states M* ∧ *q2* ∈ *states M* ∧ *q1* ≠ *q2 then the* (*Mapping.lookup distMap* (*q1*,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables M q1 q2*);
pairs = *filter* (λ (*x*,*y*) . *x* ≠ *y*) (*list-ordered-pairs* (*states-as-list M*));
distSet = *from-list* (*map* (*case-prod distHelper*) *pairs*);
distFun = (λ *M x t* . *distSet*)
in pair-framework-h-components M m distFun)
⟨*proof*⟩

lemma *w-method-via-spy-framework-code*[*code*] :
w-method-via-spy-framework M m = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
distMap = *mapping-of* (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provid*
tables M q1 q2))
(*filter* (λ *qq* . *fst qq* ≠ *snd qq*) (*List.product* (*states-as-list M*)

```
(states-as-list M))));
      distHelper = (λ q1 q2 . if q1 ∈ states M ∧ q2 ∈ states M ∧ q1 ≠ q2 then the
(Mapping.lookup distMap (q1,q2)) else get-distinguishing-sequence-from-ofsm-tables
M q1 q2);
      pairs = filter (λ (x,y) . x ≠ y) (list-ordered-pairs (states-as-list M));
      distSet = from-list (map (case-prod distHelper) pairs);
      distFun = (λ k q . distSet)
    in spy-framework-static-with-empty-graph M distFun m)
  ⟨proof⟩
```

lemma *w-method-via-h-framework-code*[*code*] :

```
  w-method-via-h-framework M m = (let
      tables = (compute-ofsm-tables M (size M − 1));
    distMap = mapping-of (map (λ (q1,q2) . ((q1,q2), get-distinguishing-sequence-from-ofsm-tables-with-provid
tables M q1 q2))
                        (filter (λ qq . fst qq ≠ snd qq) (List.product (states-as-list M)
(states-as-list M))));
      distHelper = (λ q1 q2 . if q1 ∈ states M ∧ q2 ∈ states M ∧ q1 ≠ q2 then the
(Mapping.lookup distMap (q1,q2)) else get-distinguishing-sequence-from-ofsm-tables
M q1 q2);
      pairs = filter (λ (x,y) . x ≠ y) (list-ordered-pairs (states-as-list M));
      distSet = from-list (map (case-prod distHelper) pairs);
      distFun = (λ k q . distSet)
    in h-framework-static-with-empty-graph M distFun m)
  ⟨proof⟩
```

lemma *w-method-via-h-framework-2-code*[*code*] :

```
  w-method-via-h-framework-2 M m = (let
      tables = (compute-ofsm-tables M (size M − 1));
    distMap = mapping-of (map (λ (q1,q2) . ((q1,q2), get-distinguishing-sequence-from-ofsm-tables-with-provid
tables M q1 q2))
                        (filter (λ qq . fst qq ≠ snd qq) (List.product (states-as-list M)
(states-as-list M))));
      distHelper = (λ q1 q2 . if q1 ∈ states M ∧ q2 ∈ states M ∧ q1 ≠ q2 then the
(Mapping.lookup distMap (q1,q2)) else get-distinguishing-sequence-from-ofsm-tables
M q1 q2);
      pairs = filter (λ (x,y) . x ≠ y) (list-ordered-pairs (states-as-list M));
      handlePair = (λ W (q,q') . if contains-distinguishing-trace M W q q'
                              then W
                              else insert W (distHelper q q'));
      distSet = foldl handlePair empty pairs;
      distFun = (λ k q . distSet)
    in h-framework-static-with-empty-graph M distFun m)
  ⟨proof⟩
```

end

32 Implementations of the Wp-Method

theory *Wp-Method-Implementations*
imports *Intermediate-Frameworks Pair-Framework ../Distinguishability Test-Suite-Representations ../OFSM-Tables-Refined HOL−Library.List-Lexorder*
begin

32.1 Distinguishing Sets

fun *add-distinguishing-set-or-state-identifier* :: *nat* ⇒ (*'a* :: *linorder*, *'b* :: *linorder*, *'c* :: *linorder*) *fsm* ⇒ ((*'b* × *'c*) *list* × *'a*) × (*'b* × *'c*) *list* × *'a* ⇒ (*'b* × *'c*) *prefix-tree* ⇒ (*'b*×*'c*) *prefix-tree* **where**
  add-distinguishing-set-or-state-identifier k M ((*io1*,*q1*),(*io2*,*q2*)) *t* = (*if length io1* = *k* ∨ *length io2* = *k*
    then insert empty (*get-distinguishing-sequence-from-ofsm-tables M q1 q2*)
    else distinguishing-set M)

lemma *add-distinguishing-set-or-state-identifier-distinguishes* :
  assumes *observable M*
  and *minimal M*
  and $\alpha \in L\ M$
  and $\beta \in L\ M$
  and *after-initial M* $\alpha \neq$ *after-initial M* β
shows ∃ *io* ∈ *set* (*add-distinguishing-set-or-state-identifier k M* ((α,*after-initial M* α),(β,*after-initial M* β)) *t*) ∪ (*set* (*after t* α) ∩ *set* (*after t* β)) . *distinguishes M* (*after-initial M* α) (*after-initial M* β) *io*
⟨*proof*⟩

lemma *add-distinguishing-set-or-state-identifier-finite* :
 finite-tree ((*add-distinguishing-set-or-state-identifier k*) *M* ((α,*after-initial M* α),(β,*after-initial M* β)) *t*)
⟨*proof*⟩

fun *distinguishing-set-or-state-identifier* :: *nat* ⇒ (*'a* :: *linorder*, *'b* :: *linorder*, *'c* :: *linorder*) *fsm* ⇒ *nat* ⇒ *'a* ⇒ (*'b*×*'c*) *prefix-tree* **where**
  distinguishing-set-or-state-identifier l M k q = (*if k* = *l*
    then get-HSI M q
    else distinguishing-set M)

lemma *get-HSI-subset* :
  assumes *observable M*
  and *minimal M*
  and *q* ∈ *states M*
shows *set* (*get-HSI M q*) ⊆ *set* (*distinguishing-set M*)
⟨*proof*⟩

lemma *distinguishing-set-or-state-identifier-distinguishes* :

assumes *observable M*
and *minimal M*
and *q1 ∈ states M* **and** *q2 ∈ states M* **and** *q1 ≠ q2*
shows ∃ *io* . ∀ *k1 k2* . *io* ∈ *set* (*distinguishing-set-or-state-identifier l M k1 q1*) ∩ *set* (*distinguishing-set-or-state-identifier l M k2 q2*) ∧ *distinguishes M q1 q2 io*
⟨*proof*⟩

lemma *distinguishing-set-or-state-identifier-finite* :
finite-tree (*distinguishing-set-or-state-identifier l M k q*)
⟨*proof*⟩

32.2 Using the H-Framework

definition *wp-method-via-h-framework* :: (*'a::linorder,'b::linorder,'c::linorder*) *fsm* ⇒ *nat* ⇒ (*'b×'c*) *prefix-tree* **where**
wp-method-via-h-framework M m = h-framework-static-with-empty-graph M (*distinguishing-set-or-state-identi*
(*Suc* (*m* − *size-r M*)) *M*) *m*

definition *wp-method-via-h-framework-lists* :: (*'a::linorder,'b::linorder,'c::linorder*) *fsm* ⇒ *nat* ⇒ ((*'b×'c*) × *bool*) *list list* **where**
wp-method-via-h-framework-lists M m = sorted-list-of-maximal-sequences-in-tree (*test-suite-from-io-tree M* (*initial M*) (*wp-method-via-h-framework M m*))

lemma *wp-method-via-h-framework-completeness-and-finiteness* :
fixes *M1* :: (*'a::linorder,'b::linorder,'c::linorder*) *fsm*
fixes *M2* :: (*'e,'b,'c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*wp-method-via-h-framework M1 m*)) = (*L M2* ∩ *set* (*wp-method-via-h-framework M1 m*)))
and *finite-tree* (*wp-method-via-h-framework M1 m*)
⟨*proof*⟩

lemma *wp-method-via-h-framework-lists-completeness* :
fixes *M1* :: (*'a::linorder,'b::linorder,'c::linorder*) *fsm*
fixes *M2* :: (*'d,'b,'c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*

and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*initial M2*)) (*wp-method-via-h-framework-lists M1 m*)
⟨*proof*⟩

32.3 Using the SPY-Framework

definition *wp-method-via-spy-framework* :: (′*a*::*linorder*,′*b*::*linorder*,′*c*::*linorder*) *fsm* ⇒ *nat* ⇒ (′*b*×′*c*) *prefix-tree* **where**
wp-method-via-spy-framework M m = *spy-framework-static-with-empty-graph M* (*distinguishing-set-or-state-identifier* (*Suc* (*m* − *size-r M*)) *M*) *m*

lemma *wp-method-via-spy-framework-completeness-and-finiteness* :
fixes *M1* :: (′*a*::*linorder*,′*b*::*linorder*,′*c*::*linorder*) *fsm*
fixes *M2* :: (′*d*,′*b*,′*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*wp-method-via-spy-framework M1 m*)) = (*L M2* ∩ *set* (*wp-method-via-spy-framework M1 m*)))
and *finite-tree* (*wp-method-via-spy-framework M1 m*)
⟨*proof*⟩

definition *wp-method-via-spy-framework-lists* :: (′*a*::*linorder*,′*b*::*linorder*,′*c*::*linorder*) *fsm* ⇒ *nat* ⇒ ((′*b*×′*c*) × *bool*) *list list* **where**
wp-method-via-spy-framework-lists M m = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*wp-method-via-spy-framework M m*))

lemma *wp-method-via-spy-framework-lists-completeness* :
fixes *M1* :: (′*a*::*linorder*,′*b*::*linorder*,′*c*::*linorder*) *fsm*
fixes *M2* :: (′*d*,′*b*,′*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*initial M2*)) (*wp-method-via-spy-framework-lists M1 m*)
⟨*proof*⟩

32.4 Code Generation

lemma *wp-method-via-spy-framework-code*[*code*] :
wp-method-via-spy-framework M m = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
distMap = *mapping-of* (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provid*
tables M q1 q2))
(*filter* (λ *qq* . *fst qq* ≠ *snd qq*) (*List.product* (*states-as-list M*)
(*states-as-list M*))));
distHelper = (λ *q1 q2* . *if q1* ∈ *states M* ∧ *q2* ∈ *states M* ∧ *q1* ≠ *q2 then the*
(*Mapping.lookup distMap* (*q1*,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables*
M q1 q2);
pairs = *filter* (λ (*x*,*y*) . *x* ≠ *y*) (*list-ordered-pairs* (*states-as-list M*));
distSet = *from-list* (*map* (*case-prod distHelper*) *pairs*);
hsiMap = *mapping-of* (*map* (λ *q* . (*q*,*from-list* (*map* (λ*q'* . *distHelper q q'*)
(*filter* ((≠) *q*) (*states-as-list M*))))) (*states-as-list M*));
l = (*Suc* (*m* − *size-r M*));
distFun = (λ *k q* . *if k* = *l*
then (*if q* ∈ *states M then the* (*Mapping.lookup hsiMap q*) *else*
get-HSI M q)
else distSet)
in spy-framework-static-with-empty-graph M distFun m)
(**is** *?f1* = *?f2*)
⟨*proof*⟩

lemma *wp-method-via-h-framework-code*[*code*] :
wp-method-via-h-framework M m = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
distMap = *mapping-of* (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provid*
tables M q1 q2))
(*filter* (λ *qq* . *fst qq* ≠ *snd qq*) (*List.product* (*states-as-list M*)
(*states-as-list M*))));
distHelper = (λ *q1 q2* . *if q1* ∈ *states M* ∧ *q2* ∈ *states M* ∧ *q1* ≠ *q2 then the*
(*Mapping.lookup distMap* (*q1*,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables*
M q1 q2);
pairs = *filter* (λ (*x*,*y*) . *x* ≠ *y*) (*list-ordered-pairs* (*states-as-list M*));
distSet = *from-list* (*map* (*case-prod distHelper*) *pairs*);
hsiMap = *mapping-of* (*map* (λ *q* . (*q*,*from-list* (*map* (λ*q'* . *distHelper q q'*)
(*filter* ((≠) *q*) (*states-as-list M*))))) (*states-as-list M*));
l = (*Suc* (*m* − *size-r M*));
distFun = (λ *k q* . *if k* = *l*
then (*if q* ∈ *states M then the* (*Mapping.lookup hsiMap q*) *else*
get-HSI M q)
else distSet)
in h-framework-static-with-empty-graph M distFun m)
(**is** *?f1* = *?f2*)
⟨*proof*⟩

end

33 Backwards Reachability Analysis

This theory introduces function *select-inputs* which is used for the calculation of both state preambles and state separators.

theory *Backwards-Reachability-Analysis*
imports *../FSM*
begin

Function *select-inputs* calculates an associative list that maps states to a single input each such that the FSM induced by this input selection is acyclic, single input and whose only deadlock states (if any) are contained in *stateSet*. The following parameters are used: 1) transition function *f* (typically (*h M*) for some FSM *M*) 2) a source state *q0* (selection terminates as soon as this states is assigned some input) 3) a list of inputs that may be assigned to states 4) a list of states not yet taken (these are considered when searching for the next possible assignment) 5) a set *stateSet* of all states that already have an input assigned to them by *m* 6) an associative list *m* containing previously chosen assignments

function *select-inputs* :: $(('a \times 'b) \Rightarrow ('c \times 'a)\ set) \Rightarrow 'a \Rightarrow 'b\ list \Rightarrow 'a\ list \Rightarrow 'a\ set \Rightarrow ('a \times 'b)\ list \Rightarrow ('a \times 'b)\ list$ **where**
select-inputs f q0 inputList [] *stateSet m* = (*case find* (λ *x* . *f* (*q0*,*x*) ≠ {} ∧ (∀ (*y*,*q''*) ∈ *f* (*q0*,*x*) . (*q''* ∈ *stateSet*))) *inputList of*
Some x ⇒ *m*@[(*q0*,*x*)] |
None ⇒ *m*) |
select-inputs f q0 inputList (*n*#*nL*) *stateSet m* =
(*case find* (λ *x* . *f* (*q0*,*x*) ≠ {} ∧ (∀ (*y*,*q''*) ∈ *f* (*q0*,*x*) . (*q''* ∈ *stateSet*))) *inputList of*
Some x ⇒ *m*@[(*q0*,*x*)] |
None ⇒ (*case find-remove-2* (λ *q' x* . *f* (*q'*,*x*) ≠ {} ∧ (∀ (*y*,*q''*) ∈ *f* (*q'*,*x*) . (*q''* ∈ *stateSet*))) (*n*#*nL*) *inputList*
of None ⇒ *m* |
Some (*q'*,*x*,*stateList'*) ⇒ *select-inputs f q0 inputList stateList'* (*insert q' stateSet*) (*m*@[(*q'*,*x*)])))
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *select-inputs-length* :
length (*select-inputs f q0 inputList stateList stateSet m*) ≤ (*length m*) + *Suc* (*length stateList*)
⟨*proof*⟩

lemma *select-inputs-length-min* :
length (*select-inputs f q0 inputList stateList stateSet m*) ≥ (*length m*)
⟨*proof*⟩

lemma *select-inputs-helper1* :
  find ($\lambda x.$ *f* (*q0*, *x*) $\neq$ {} $\wedge$ ($\forall$ (*y*, *q''*)$\in$*f* (*q0*, *x*). *q''* $\in$ *nS*)) *iL* = *Some x*
    $\Longrightarrow$ (*select-inputs f q0 iL nL nS m*) = *m*@[(*q0*,*x*)]
  ⟨*proof*⟩

lemma *select-inputs-take* :
  take (*length m*) (*select-inputs f q0 inputList stateList stateSet m*) = *m*
⟨*proof*⟩

lemma *select-inputs-take'* :
  take (*length m*) (*select-inputs f q0 iL nL nS* (*m*@*m'*)) = *m*
  ⟨*proof*⟩

lemma *select-inputs-distinct* :
  assumes *distinct* (*map fst m*)
  and *set* (*map fst m*) $\subseteq$ *nS*
  and *q0* $\notin$ *nS*
  and *distinct nL*
  and *q0* $\notin$ *set nL*
  and *set nL* $\cap$ *nS* = {}
  shows *distinct* (*map fst* (*select-inputs f q0 iL nL nS m*))
⟨*proof*⟩

lemma *select-inputs-index-properties* :
  assumes *i* < *length* (*select-inputs* (*h M*) *q0 iL nL nS m*)
  and *i* $\geq$ *length m*
  and *distinct* (*map fst m*)
  and *nS* = *nS0* $\cup$ *set* (*map fst m*)
  and *q0* $\notin$ *nS*
  and *distinct nL*
  and *q0* $\notin$ *set nL*
  and *set nL* $\cap$ *nS* = {}
shows *fst* (*select-inputs* (*h M*) *q0 iL nL nS m* ! *i*) $\in$ (*insert q0* (*set nL*))
    fst (*select-inputs* (*h M*) *q0 iL nL nS m* ! *i*) $\notin$ *nS0*
    snd (*select-inputs* (*h M*) *q0 iL nL nS m* ! *i*) $\in$ *set iL*
    ($\forall$ *qx'* $\in$ *set* (*take i* (*select-inputs* (*h M*) *q0 iL nL nS m*)) . *fst* (*select-inputs* (*h M*) *q0 iL nL nS m* ! *i*) $\neq$ *fst qx'*)
    ($\exists$ *t* $\in$ *transitions M* . *t-source t* = *fst* (*select-inputs* (*h M*) *q0 iL nL nS m* ! *i*) $\wedge$ *t-input t* = *snd* (*select-inputs* (*h M*) *q0 iL nL nS m* ! *i*))
    ($\forall$ *t* $\in$ *transitions M* . (*t-source t* = *fst* (*select-inputs* (*h M*) *q0 iL nL nS m* ! *i*) $\wedge$ *t-input t* = *snd* (*select-inputs* (*h M*) *q0 iL nL nS m* ! *i*)) $\longrightarrow$ (*t-target t* $\in$ *nS0* $\vee$ ($\exists$ *qx'* $\in$ *set* (*take i* (*select-inputs* (*h M*) *q0 iL nL nS m*)) . *fst qx'* = (*t-target t*))))

⟨*proof*⟩

lemma *select-inputs-initial* :
assumes $qx \in set\ (select\text{-}inputs\ f\ q0\ iL\ nL\ nS\ m) - set\ m$
and $fst\ qx = q0$
shows $(last\ (select\text{-}inputs\ f\ q0\ iL\ nL\ nS\ m)) = qx$
⟨*proof*⟩

lemma *select-inputs-max-length* :
assumes *distinct nL*
shows $length\ (select\text{-}inputs\ f\ q0\ iL\ nL\ nS\ m) \leq length\ m + Suc\ (length\ nL)$
⟨*proof*⟩

lemma *select-inputs-q0-containment* :
assumes $f\ (q0,x) \neq \{\}$
and $(\forall\ (y,q'') \in f\ (q0,x)\ .\ (q'' \in nS))$
and $x \in set\ iL$
shows $(\exists\ qx \in set\ (select\text{-}inputs\ f\ q0\ iL\ nL\ nS\ m)\ .\ fst\ qx = q0)$
⟨*proof*⟩

lemma *select-inputs-from-submachine* :
assumes *single-input S*
and *acyclic S*
and *is-submachine S M*
and $\bigwedge\ q\ x\ .\ q \in reachable\text{-}states\ S \Longrightarrow h\ S\ (q,x) \neq \{\} \Longrightarrow h\ S\ (q,x) = h\ M\ (q,x)$
and $\bigwedge\ q\ .\ q \in reachable\text{-}states\ S \Longrightarrow deadlock\text{-}state\ S\ q \Longrightarrow q \in nS0 \cup set\ (map\ fst\ m)$
and $states\ M = insert\ (initial\ S)\ (set\ nL \cup nS0 \cup set\ (map\ fst\ m))$
and $(initial\ S) \notin (set\ nL \cup nS0 \cup set\ (map\ fst\ m))$
shows $fst\ (last\ (select\text{-}inputs\ (h\ M)\ (initial\ S)\ (inputs\text{-}as\text{-}list\ M)\ nL\ (nS0 \cup set\ (map\ fst\ m))\ m)) = (initial\ S)$
and $length\ (select\text{-}inputs\ (h\ M)\ (initial\ S)\ (inputs\text{-}as\text{-}list\ M)\ nL\ (nS0 \cup set\ (map\ fst\ m))\ m) > 0$
⟨*proof*⟩

end

34 State Separators

This theory defined state separators. A state separator *S* of some pair of states *q1*, *q2* of some FSM *M* is an acyclic single-input FSM based on the product machine *P* of *M* with initial state *q1* and *M* with initial state *q2* such that every maximal length sequence in the language of S is either in the language of *q1* or the language of *q2*, but not both. That is, C represents a strategy of distinguishing *q1* and *q2* in every complete submachine of P. In testing, separators are used to distinguish states reached in the SUT to establish a lower bound on the number of distinct states in the SUT.

theory *State-Separator*
imports *../Product-FSM Backwards-Reachability-Analysis*
begin

34.1 Canonical Separators

34.1.1 Construction

fun *canonical-separator* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $'a \Rightarrow 'a \Rightarrow (('a \times 'a) + 'a,'b,'c)$ *fsm*
where
canonical-separator M q1 q2 = (*canonical-separator' M* ((*product* (*from-FSM M q1*) (*from-FSM M q2*))) *q1 q2*)

lemma *canonical-separator-simps* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows *initial* (*canonical-separator M q1 q2*) = *Inl* (*q1*,*q2*)
states (*canonical-separator M q1 q2*)
= (*image Inl* (*states* (*product* (*from-FSM M q1*) (*from-FSM M q2*)))) ∪ {*Inr q1*, *Inr q2*}
inputs (*canonical-separator M q1 q2*) = *inputs M*
outputs (*canonical-separator M q1 q2*) = *outputs M*
transitions (*canonical-separator M q1 q2*)
= *shifted-transitions* (*transitions* ((*product* (*from-FSM M q1*) (*from-FSM M q2*))))
∪ *distinguishing-transitions* (*h-out M*) *q1 q2* (*states* ((*product* (*from-FSM M q1*) (*from-FSM M q2*)))) (*inputs* ((*product* (*from-FSM M q1*) (*from-FSM M q2*))))
⟨*proof*⟩

lemma *distinguishing-transitions-alt-def* :
distinguishing-transitions (*h-out M*) *q1 q2 PS* (*inputs M*) =
{(*Inl* (*q1'*,*q2'*),*x*,*y*,*Inr q1*) | *q1' q2' x y* . (*q1'*,*q2'*) ∈ *PS* ∧ (∃ *q'* . (*q1'*,*x*,*y*,*q'*) ∈ *transitions M*) ∧ ¬(∃ *q'* . (*q2'*,*x*,*y*,*q'*) ∈ *transitions M*)}
∪ {(*Inl* (*q1'*,*q2'*),*x*,*y*,*Inr q2*) | *q1' q2' x y* . (*q1'*,*q2'*) ∈ *PS* ∧ ¬(∃ *q'* . (*q1'*,*x*,*y*,*q'*) ∈ *transitions M*) ∧ (∃ *q'* . (*q2'*,*x*,*y*,*q'*) ∈ *transitions M*)}
(**is** *?dts* = *?dl* ∪ *?dr*)
⟨*proof*⟩

lemma *distinguishing-transitions-alt-alt-def* :
distinguishing-transitions (*h-out M*) *q1 q2 PS* (*inputs M*) =
{ *t* . ∃ *q1' q2'* . *t-source t* = *Inl* (*q1'*,*q2'*) ∧ (*q1'*,*q2'*) ∈ *PS* ∧ *t-target t* = *Inr*

q1 ∧ (∃ *t′* ∈ *transitions M* . *t-source t′* = *q1′* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*) ∧ ¬(∃ *t′* ∈ *transitions M* . *t-source t′* = *q2′* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*)}
∪ { *t* . ∃ *q1′ q2′* . *t-source t* = *Inl* (*q1′*,*q2′*) ∧ (*q1′*,*q2′*) ∈ *PS* ∧ *t-target t* = *Inr q2* ∧ ¬(∃ *t′* ∈ *transitions M* . *t-source t′* = *q1′* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*) ∧ (∃ *t′* ∈ *transitions M* . *t-source t′* = *q2′* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*)}

⟨*proof*⟩

lemma *shifted-transitions-alt-def* :
shifted-transitions ts = {(*Inl* (*q1′*,*q2′*), *x*, *y*, (*Inl* (*q1″*,*q2″*))) | *q1′ q2′ x y q1″ q2″* . ((*q1′*,*q2′*), *x*, *y*, (*q1″*,*q2″*)) ∈ *ts*}
⟨*proof*⟩

lemma *canonical-separator-transitions-helper* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows *transitions* (*canonical-separator M q1 q2*) =
(*shifted-transitions* (*transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*))))
∪ {(*Inl* (*q1′*,*q2′*),*x*,*y*,*Inr q1*) | *q1′ q2′ x y* . (*q1′*,*q2′*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) ∧ (∃ *q′* . (*q1′*,*x*,*y*,*q′*) ∈ *transitions M*) ∧ ¬(∃ *q′* . (*q2′*,*x*,*y*,*q′*) ∈ *transitions M*)}
∪ {(*Inl* (*q1′*,*q2′*),*x*,*y*,*Inr q2*) | *q1′ q2′ x y* . (*q1′*,*q2′*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) ∧ ¬(∃ *q′* . (*q1′*,*x*,*y*,*q′*) ∈ *transitions M*) ∧ (∃ *q′* . (*q2′*,*x*,*y*,*q′*) ∈ *transitions M*)}
⟨*proof*⟩

definition *distinguishing-transitions-left* :: (′*a*, ′*b*, ′*c*) *fsm* ⇒ ′*a* ⇒ ′*a* ⇒ ((′*a* × ′*a* + ′*a*) × ′*b* × ′*c* × (′*a* × ′*a* + ′*a*)) *set* **where**
distinguishing-transitions-left M q1 q2 ≡ {(*Inl* (*q1′*,*q2′*),*x*,*y*,*Inr q1*) | *q1′ q2′ x y* . (*q1′*,*q2′*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) ∧ (∃ *q′* . (*q1′*,*x*,*y*,*q′*) ∈ *transitions M*) ∧ ¬(∃ *q′* . (*q2′*,*x*,*y*,*q′*) ∈ *transitions M*)}
definition *distinguishing-transitions-right* :: (′*a*, ′*b*, ′*c*) *fsm* ⇒ ′*a* ⇒ ′*a* ⇒ ((′*a* × ′*a* + ′*a*) × ′*b* × ′*c* × (′*a* × ′*a* + ′*a*)) *set* **where**
distinguishing-transitions-right M q1 q2 ≡ {(*Inl* (*q1′*,*q2′*),*x*,*y*,*Inr q2*) | *q1′ q2′ x y* . (*q1′*,*q2′*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) ∧ ¬(∃ *q′* . (*q1′*,*x*,*y*,*q′*) ∈ *transitions M*) ∧ (∃ *q′* . (*q2′*,*x*,*y*,*q′*) ∈ *transitions M*)}

definition *distinguishing-transitions-left-alt* :: (′*a*, ′*b*, ′*c*) *fsm* ⇒ ′*a* ⇒ ′*a* ⇒ ((′*a* × ′*a* + ′*a*) × ′*b* × ′*c* × (′*a* × ′*a* + ′*a*)) *set* **where**
distinguishing-transitions-left-alt M q1 q2 ≡ { *t* . ∃ *q1′ q2′* . *t-source t* = *Inl* (*q1′*,*q2′*) ∧ (*q1′*,*q2′*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) ∧ *t-target t* = *Inr q1* ∧ (∃ *t′* ∈ *transitions M* . *t-source t′* = *q1′* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*) ∧ ¬(∃ *t′* ∈ *transitions M* . *t-source t′* = *q2′* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*)}

definition *distinguishing-transitions-right-alt* :: ('a, 'b, 'c) *fsm* ⇒ 'a ⇒ 'a ⇒ (('a × 'a + 'a) × 'b × 'c × ('a × 'a + 'a)) *set* **where**
distinguishing-transitions-right-alt M q1 q2 ≡ { *t* . ∃ *q1' q2'* . *t-source t* = *Inl* (*q1'*,*q2'*) ∧ (*q1'*,*q2'*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) ∧ *t-target t* = *Inr q2* ∧ ¬(∃ *t'* ∈ *transitions M* . *t-source t'* = *q1'* ∧ *t-input t'* = *t-input t* ∧ *t-output t'* = *t-output t*) ∧ (∃ *t'* ∈ *transitions M* . *t-source t'* = *q2'* ∧ *t-input t'* = *t-input t* ∧ *t-output t'* = *t-output t*)}

definition *shifted-transitions-for* :: ('a, 'b, 'c) *fsm* ⇒ 'a ⇒ 'a ⇒ (('a × 'a + 'a) × 'b × 'c × ('a × 'a + 'a)) *set* **where**
shifted-transitions-for M q1 q2 ≡ {(*Inl* (*t-source t*),*t-input t*, *t-output t*, *Inl* (*t-target t*)) | *t* . *t* ∈ *transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*))}

lemma *shifted-transitions-for-alt-def* :
shifted-transitions-for M q1 q2 = {(*Inl* (*q1'*,*q2'*), *x*, *y*, (*Inl* (*q1''*,*q2''*))) | *q1' q2' x y q1'' q2''* . ((*q1'*,*q2'*), *x*, *y*, (*q1''*,*q2''*)) ∈ *transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*))}
⟨*proof*⟩

lemma *distinguishing-transitions-left-alt-alt-def* :
distinguishing-transitions-left M q1 q2 = *distinguishing-transitions-left-alt M q1 q2*
⟨*proof*⟩

lemma *distinguishing-transitions-right-alt-alt-def* :
distinguishing-transitions-right M q1 q2 = *distinguishing-transitions-right-alt M q1 q2*
⟨*proof*⟩

lemma *canonical-separator-transitions-def* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows *transitions* (*canonical-separator M q1 q2*) =
{(*Inl* (*q1'*,*q2'*), *x*, *y*, (*Inl* (*q1''*,*q2''*))) | *q1' q2' x y q1'' q2''* . ((*q1'*,*q2'*), *x*, *y*, (*q1''*,*q2''*)) ∈ *transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*))}
∪ (*distinguishing-transitions-left M q1 q2*)
∪ (*distinguishing-transitions-right M q1 q2*)
⟨*proof*⟩

lemma *canonical-separator-transitions-alt-def* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows *transitions* (*canonical-separator M q1 q2*) =
(*shifted-transitions-for M q1 q2*)
∪ (*distinguishing-transitions-left-alt M q1 q2*)
∪ (*distinguishing-transitions-right-alt M q1 q2*)

⟨*proof*⟩

34.1.2 State Separators as Submachines of Canonical Separators

definition *is-state-separator-from-canonical-separator* :: (('a × 'a) + 'a, 'b, 'c) *fsm* ⇒ 'a ⇒ 'a ⇒ (('a × 'a) + 'a, 'b, 'c) *fsm* ⇒ *bool* **where**
is-state-separator-from-canonical-separator CSep q1 q2 S = (
is-submachine S CSep
∧ *single-input S*
∧ *acyclic S*
∧ *deadlock-state S* (*Inr q1*)
∧ *deadlock-state S* (*Inr q2*)
∧ ((*Inr q1*) ∈ *reachable-states S*)
∧ ((*Inr q2*) ∈ *reachable-states S*)
∧ (∀ *q* ∈ *reachable-states S* . (*q* ≠ *Inr q1* ∧ *q* ≠ *Inr q2*) ⟶ (*isl q* ∧ ¬ *deadlock-state S q*))
∧ (∀ *q* ∈ *reachable-states S* . ∀ *x* ∈ (*inputs CSep*) . (∃ *t* ∈ *transitions S* . *t-source t* = *q* ∧ *t-input t* = *x*) ⟶ (∀ *t'* ∈ *transitions CSep* . *t-source t'* = *q* ∧ *t-input t'* = *x* ⟶ *t'* ∈ *transitions S*))
)

34.1.3 Canonical Separator Properties

lemma *is-state-separator-from-canonical-separator-simps* :
assumes *is-state-separator-from-canonical-separator CSep q1 q2 S*
shows *is-submachine S CSep*
and *single-input S*
and *acyclic S*
and *deadlock-state S* (*Inr q1*)
and *deadlock-state S* (*Inr q2*)
and ((*Inr q1*) ∈ *reachable-states S*)
and ((*Inr q2*) ∈ *reachable-states S*)
and ⋀ *q* . *q* ∈ *reachable-states S* ⟹ *q* ≠ *Inr q1* ⟹ *q* ≠ *Inr q2* ⟹ (*isl q* ∧ ¬ *deadlock-state S q*)
and ⋀ *q x t* . *q* ∈ *reachable-states S* ⟹ *x* ∈ (*inputs CSep*) ⟹ (∃ *t* ∈ *transitions S* . *t-source t* = *q* ∧ *t-input t* = *x*) ⟹ *t* ∈ *transitions CSep* ⟹ *t-source t* = *q* ⟹ *t-input t* = *x* ⟹ *t* ∈ *transitions S*
⟨*proof*⟩

lemma *is-state-separator-from-canonical-separator-initial* :
assumes *is-state-separator-from-canonical-separator* (*canonical-separator M q1 q2*) *q1 q2 A*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *initial A* = *Inl* (*q1*,*q2*)
⟨*proof*⟩

lemma *path-shift-Inl* :

assumes (*image shift-Inl* (*transitions M*)) ⊆ (*transitions C*)
and ⋀ *t* . *t* ∈ (*transitions C*) ⟹ *isl* (*t-target t*) ⟹ ∃ *t'* ∈ *transitions M* . *t* = (*Inl* (*t-source t'*), *t-input t'*, *t-output t'*, *Inl* (*t-target t'*))
and *initial C* = *Inl* (*initial M*)
and (*inputs C*) = (*inputs M*)
and (*outputs C*) = (*outputs M*)
shows *path M* (*initial M*) *p* = *path C* (*initial C*) (*map shift-Inl p*)
⟨*proof*⟩

lemma *canonical-separator-product-transitions-subset* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows *image shift-Inl* (*transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*))) ⊆ (*transitions* (*canonical-separator M q1 q2*))
⟨*proof*⟩

lemma *canonical-separator-transition-targets* :
assumes *t* ∈ (*transitions* (*canonical-separator M q1 q2*))
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *isl* (*t-target t*) ⟹ *t* ∈ {(*Inl* (*t-source t*),*t-input t*, *t-output t*, *Inl* (*t-target t*)) | *t* . *t* ∈ *transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*))}
and *t-target t* = *Inr q1* ⟹ *q1* ≠ *q2* ⟹ *t* ∈ (*distinguishing-transitions-left-alt M q1 q2*)
and *t-target t* = *Inr q2* ⟹ *q1* ≠ *q2* ⟹ *t* ∈ (*distinguishing-transitions-right-alt M q1 q2*)
and *isl* (*t-target t*) ∨ *t-target t* = *Inr q1* ∨ *t-target t* = *Inr q2*
⟨*proof*⟩

lemma *canonical-separator-path-shift* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows *path* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*initial* (*product* (*from-FSM M q1*) (*from-FSM M q2*))) *p*
= *path* (*canonical-separator M q1 q2*) (*initial* (*canonical-separator M q1 q2*)) (*map shift-Inl p*)
⟨*proof*⟩

lemma *canonical-separator-t-source-isl* :
assumes *t* ∈ (*transitions* (*canonical-separator M q1 q2*))
and *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows *isl* (*t-source t*)
⟨*proof*⟩

lemma *canonical-separator-path-from-shift* :
assumes *path* (*canonical-separator M q1 q2*) (*initial* (*canonical-separator M q1

q2)) *p*
  and *isl* (*target* (*initial* (*canonical-separator M q1 q2*)) *p*)
  and *q1* ∈ *states M* **and** *q2* ∈ *states M*
  shows ∃ *p′* . *path* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*initial* (*product* (*from-FSM M q1*) (*from-FSM M q2*))) *p′*
  ∧ *p* = (*map shift-Inl p′*)
⟨*proof*⟩

lemma *shifted-transitions-targets* :
  assumes *t* ∈ (*shifted-transitions ts*)
  shows *isl* (*t-target t*)
  ⟨*proof*⟩

lemma *distinguishing-transitions-left-sources-targets* :
  assumes *t* ∈ (*distinguishing-transitions-left-alt M q1 q2*)
  and *q2* ∈ *states M*
  obtains *q1′ q2′ t′* **where** *t-source t* = *Inl* (*q1′*,*q2′*)
  q1′ ∈ *states M*
  q2′ ∈ *states M*
  t′ ∈ *transitions M*
  t-source t′ = *q1′*
  t-input t′ = *t-input t*
  t-output t′ = *t-output t*
  ¬ (∃ *t″*∈ *transitions M*. *t-source t″* = *q2′* ∧ *t-input t″* = *t-input t* ∧ *t-output t″* = *t-output t*)
  t-target t = *Inr q1*
  ⟨*proof*⟩

lemma *distinguishing-transitions-right-sources-targets* :
  assumes *t* ∈ (*distinguishing-transitions-right-alt M q1 q2*)
  and *q1* ∈ *states M*
  obtains *q1′ q2′ t′* **where** *t-source t* = *Inl* (*q1′*,*q2′*)
  q1′ ∈ *states M*
  q2′ ∈ *states M*
  t′ ∈ *transitions M*
  t-source t′ = *q2′*
  t-input t′ = *t-input t*
  t-output t′ = *t-output t*
  ¬ (∃ *t″*∈ *transitions M*. *t-source t″* = *q1′* ∧ *t-input t″* = *t-input t* ∧ *t-output t″* = *t-output t*)
  t-target t = *Inr q2*
  ⟨*proof*⟩

lemma *product-from-transition-split* :
  assumes *t* ∈ *transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*))

and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows (∃ *t*′∈ *transitions M*. *t-source t*′ = *fst* (*t-source t*) ∧ *t-input t*′ = *t-input t* ∧ *t-output t*′ = *t-output t*)
and (∃ *t*′∈ *transitions M*. *t-source t*′ = *snd* (*t-source t*) ∧ *t-input t*′ = *t-input t* ∧ *t-output t*′ = *t-output t*)
⟨*proof*⟩

lemma *shifted-transitions-underlying-transition* :
assumes *tS* ∈ *shifted-transitions-for M q1 q2*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
obtains *t* **where** *tS* = (*Inl* (*t-source t*), *t-input t*, *t-output t*, *Inl* (*t-target t*))
and *t* ∈ (*transitions* ((*product* (*from-FSM M q1*) (*from-FSM M q2*))))
and (∃ *t*′∈(*transitions M*).
t-source t′ = *fst* (*t-source t*) ∧
t-input t′ = *t-input t* ∧ *t-output t*′ = *t-output t*)
and (∃ *t*′∈(*transitions M*).
t-source t′ = *snd* (*t-source t*) ∧
t-input t′ = *t-input t* ∧ *t-output t*′ = *t-output t*)
⟨*proof*⟩

lemma *shifted-transitions-observable-against-distinguishing-transitions-left* :
assumes *t1* ∈ (*shifted-transitions-for M q1 q2*)
and *t2* ∈ (*distinguishing-transitions-left M q1 q2*)
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows ¬ (*t-source t1* = *t-source t2* ∧ *t-input t1* = *t-input t2* ∧ *t-output t1* = *t-output t2*)
⟨*proof*⟩

lemma *shifted-transitions-observable-against-distinguishing-transitions-right* :
assumes *t1* ∈ (*shifted-transitions-for M q1 q2*)
and *t2* ∈ (*distinguishing-transitions-right M q1 q2*)
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows ¬ (*t-source t1* = *t-source t2* ∧ *t-input t1* = *t-input t2* ∧ *t-output t1* = *t-output t2*)
⟨*proof*⟩

lemma *distinguishing-transitions-left-observable-against-distinguishing-transitions-right* :
assumes *t1* ∈ (*distinguishing-transitions-left M q1 q2*)
and *t2* ∈ (*distinguishing-transitions-right M q1 q2*)
shows ¬ (*t-source t1* = *t-source t2* ∧ *t-input t1* = *t-input t2* ∧ *t-output t1* = *t-output t2*)

⟨*proof*⟩

lemma *distinguishing-transitions-left-observable-against-distinguishing-transitions-left* :
assumes *t1* ∈ (*distinguishing-transitions-left M q1 q2*)
and *t2* ∈ (*distinguishing-transitions-left M q1 q2*)
and *t-source t1* = *t-source t2* ∧ *t-input t1* = *t-input t2* ∧ *t-output t1* = *t-output t2*
shows *t1* = *t2*
⟨*proof*⟩

lemma *distinguishing-transitions-right-observable-against-distinguishing-transitions-right* :
assumes *t1* ∈ (*distinguishing-transitions-right M q1 q2*)
and *t2* ∈ (*distinguishing-transitions-right M q1 q2*)
and *t-source t1* = *t-source t2* ∧ *t-input t1* = *t-input t2* ∧ *t-output t1* = *t-output t2*
shows *t1* = *t2*
⟨*proof*⟩

lemma *shifted-transitions-observable-against-shifted-transitions* :
assumes *t1* ∈ (*shifted-transitions-for M q1 q2*)
and *t2* ∈ (*shifted-transitions-for M q1 q2*)
and *observable M*
and *t-source t1* = *t-source t2* ∧ *t-input t1* = *t-input t2* ∧ *t-output t1* = *t-output t2*
shows *t1* = *t2*
⟨*proof*⟩

lemma *canonical-separator-observable* :
assumes *observable M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *observable* (*canonical-separator M q1 q2*) (**is** *observable ?CSep*)
⟨*proof*⟩

lemma *canonical-separator-targets-ineq* :
assumes *t* ∈ *transitions* (*canonical-separator M q1 q2*)
and *q1* ∈ *states M* **and** *q2* ∈ *states M* **and** *q1* ≠ *q2*
shows *isl* (*t-target t*) ⟹ *t* ∈ (*shifted-transitions-for M q1 q2*)
and *t-target t* = *Inr q1* ⟹ *t* ∈ (*distinguishing-transitions-left M q1 q2*)
and *t-target t* = *Inr q2* ⟹ *t* ∈ (*distinguishing-transitions-right M q1 q2*)
⟨*proof*⟩

lemma *canonical-separator-targets-observable* :
assumes *t* ∈ *transitions* (*canonical-separator M q1 q2*)
and *q1* ∈ *states M* **and** *q2* ∈ *states M* **and** *q1* ≠ *q2*
shows *isl* (*t-target t*) ⟹ *t* ∈ (*shifted-transitions-for M q1 q2*)
and *t-target t* = *Inr q1* ⟹ *t* ∈ (*distinguishing-transitions-left M q1 q2*)
and *t-target t* = *Inr q2* ⟹ *t* ∈ (*distinguishing-transitions-right M q1 q2*)
⟨*proof*⟩

lemma *canonical-separator-maximal-path-distinguishes-left* :
assumes *is-state-separator-from-canonical-separator* (*canonical-separator M q1 q2*) *q1 q2 S* (**is** *is-state-separator-from-canonical-separator ?C q1 q2 S*)
and *path S* (*initial S*) *p*
and *target* (*initial S*) *p* = *Inr q1*
and *observable M*
and *q1* ∈ *states M* **and** *q2* ∈ *states M* **and** *q1* ≠ *q2*
shows *p-io p* ∈ *LS M q1* − *LS M q2*
⟨*proof*⟩

lemma *canonical-separator-maximal-path-distinguishes-right* :
assumes *is-state-separator-from-canonical-separator* (*canonical-separator M q1 q2*) *q1 q2 S*
(**is** *is-state-separator-from-canonical-separator ?C q1 q2 S*)
and *path S* (*initial S*) *p*
and *target* (*initial S*) *p* = *Inr q2*
and *observable M*
and *q1* ∈ *states M* **and** *q2* ∈ *states M* **and** *q1* ≠ *q2*
shows *p-io p* ∈ *LS M q2* − *LS M q1*
⟨*proof*⟩

lemma *state-separator-from-canonical-separator-observable* :
assumes *is-state-separator-from-canonical-separator* (*canonical-separator M q1 q2*) *q1 q2 A*
and *observable M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *observable A*
⟨*proof*⟩

lemma *canonical-separator-initial* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows *initial* (*canonical-separator M q1 q2*) = *Inl* (*q1*,*q2*)
⟨*proof*⟩

lemma *canonical-separator-states* :
  assumes *Inl* (*s1*,*s2*) ∈ *states* (*canonical-separator M q1 q2*)
  and *q1* ∈ *states M*
  and *q2* ∈ *states M*
shows (*s1*,*s2*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
  ⟨*proof*⟩

lemma *canonical-separator-transition* :
  assumes *t* ∈ *transitions* (*canonical-separator M q1 q2*) (**is** *t* ∈ *transitions ?C*)
  and *q1* ∈ *states M*
  and *q2* ∈ *states M*
  and *t-source t* = *Inl* (*s1*,*s2*)
  and *observable M*
  and *q1* ≠ *q2*
shows ⋀ *s1′ s2′* . *t-target t* = *Inl* (*s1′*,*s2′*) ⟹ (*s1*, *t-input t*, *t-output t*, *s1′*) ∈ *transitions M* ∧ (*s2*, *t-input t*, *t-output t*, *s2′*) ∈ *transitions M*
and *t-target t* = *Inr q1* ⟹ (∃ *t′*∈ *transitions M* . *t-source t′* = *s1* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*)
∧ (¬(∃ *t′*∈ *transitions M* . *t-source t′* = *s2* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*))
and *t-target t* = *Inr q2* ⟹ (∃ *t′*∈ *transitions M* . *t-source t′* = *s2* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*)
∧ (¬(∃ *t′*∈ *transitions M* . *t-source t′* = *s1* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*))
and (∃ *s1′ s2′* . *t-target t* = *Inl* (*s1′*,*s2′*)) ∨ *t-target t* = *Inr q1* ∨ *t-target t* = *Inr q2*
⟨*proof*⟩

lemma *canonical-separator-transition-source* :
  assumes *t* ∈ *transitions* (*canonical-separator M q1 q2*) (**is** *t* ∈ *transitions ?C*)
  and *q1* ∈ *states M*
  and *q2* ∈ *states M*
obtains *q1′ q2′* **where** *t-source t* = *Inl* (*q1′*,*q2′*)
(*q1′*,*q2′*) ∈ *states* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*))
⟨*proof*⟩

lemma *canonical-separator-transition-ex* :
  assumes *t* ∈ *transitions* (*canonical-separator M q1 q2*) (**is** *t* ∈ *transitions ?C*)
  and *q1* ∈ *states M*
  and *q2* ∈ *states M*
  and *t-source t* = *Inl* (*s1*,*s2*)
shows (∃ *t1* ∈ *transitions M* . *t-source t1* = *s1* ∧ *t-input t1* = *t-input t* ∧ *t-output t1* = *t-output t*) ∨
(∃ *t2* ∈ *transitions M* . *t-source t2* = *s2* ∧ *t-input t2* = *t-input t* ∧ *t-output t2* = *t-output t*)

⟨proof⟩

lemma *canonical-separator-path-split-target-isl* :
assumes *path* (*canonical-separator M q1 q2*) (*initial* (*canonical-separator M q1 q2*)) (*p*@[*t*])
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *isl* (*target* (*initial* (*canonical-separator M q1 q2*)) *p*)
⟨proof⟩

lemma *canonical-separator-path-initial* :
assumes *path* (*canonical-separator M q1 q2*) (*initial* (*canonical-separator M q1 q2*)) *p* (**is** *path ?C* (*initial ?C*) *p*)
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *observable M*
and *q1* ≠ *q2*
shows ⋀ *s1' s2'* . *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inl* (*s1'*,*s2'*) ⟹ (∃ *p1 p2* . *path M q1 p1* ∧ *path M q2 p2* ∧ *p-io p1* = *p-io p2* ∧ *p-io p1* = *p-io p* ∧ *target q1 p1* = *s1'* ∧ *target q2 p2* = *s2'*)
and *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inr q1* ⟹ (∃ *p1 p2 t* . *path M q1* (*p1*@[*t*]) ∧ *path M q2 p2* ∧ *p-io* (*p1*@[*t*]) = *p-io p* ∧ *p-io p2* = *butlast* (*p-io p*)) ∧ (¬(∃ *p2* . *path M q2 p2* ∧ *p-io p2* = *p-io p*))
and *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inr q2* ⟹ (∃ *p1 p2 t* . *path M q1 p1* ∧ *path M q2* (*p2*@[*t*]) ∧ *p-io p1* = *butlast* (*p-io p*) ∧ *p-io* (*p2*@[*t*]) = *p-io p*) ∧ (¬(∃ *p1* . *path M q1 p1* ∧ *p-io p1* = *p-io p*))
and (∃ *s1' s2'* . *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inl* (*s1'*,*s2'*)) ∨ *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inr q1* ∨ *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inr q2*
⟨proof⟩

lemma *canonical-separator-path-initial-ex* :
assumes *path* (*canonical-separator M q1 q2*) (*initial* (*canonical-separator M q1 q2*)) *p* (**is** *path ?C* (*initial ?C*) *p*)
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows (∃ *p1* . *path M q1 p1* ∧ *p-io p1* = *p-io p*) ∨ (∃ *p2* . *path M q2 p2* ∧ *p-io p2* = *p-io p*)
and (∃ *p1 p2* . *path M q1 p1* ∧ *path M q2 p2* ∧ *p-io p1* = *butlast* (*p-io p*) ∧ *p-io p2* = *butlast* (*p-io p*))
⟨proof⟩

lemma *canonical-separator-language* :
assumes *q1* ∈ *states M*

and $q2 \in states\ M$
shows $L\ (canonical\text{-}separator\ M\ q1\ q2) \subseteq L\ (from\text{-}FSM\ M\ q1) \cup L\ (from\text{-}FSM\ M\ q2)$ (**is** $L\ ?C \subseteq L\ ?M1 \cup L\ ?M2$)
⟨*proof*⟩

lemma *canonical-separator-language-prefix* :
assumes $io@[xy] \in L\ (canonical\text{-}separator\ M\ q1\ q2)$
and $q1 \in states\ M$
and $q2 \in states\ M$
and *observable M*
and $q1 \neq q2$
shows $io \in LS\ M\ q1$
and $io \in LS\ M\ q2$
⟨*proof*⟩

lemma *canonical-separator-distinguishing-transitions-left-containment* :
assumes $t \in (distinguishing\text{-}transitions\text{-}left\ M\ q1\ q2)$
and $q1 \in states\ M$ **and** $q2 \in states\ M$
shows $t \in transitions\ (canonical\text{-}separator\ M\ q1\ q2)$
⟨*proof*⟩

lemma *canonical-separator-distinguishing-transitions-right-containment* :
assumes $t \in (distinguishing\text{-}transitions\text{-}right\ M\ q1\ q2)$
and $q1 \in states\ M$ **and** $q2 \in states\ M$
shows $t \in transitions\ (canonical\text{-}separator\ M\ q1\ q2)$ (**is** $t \in transitions\ ?C$)
⟨*proof*⟩

lemma *distinguishing-transitions-left-alt-intro* :
assumes $(s1,s2) \in states\ (Product\text{-}FSM.product\ (FSM.from\text{-}FSM\ M\ q1)\ (FSM.from\text{-}FSM\ M\ q2))$
and $(\exists\, t \in transitions\ M.\ t\text{-}source\ t = s1 \wedge t\text{-}input\ t = x \wedge t\text{-}output\ t = y)$
and $\neg(\exists\, t \in transitions\ M.\ t\text{-}source\ t = s2 \wedge t\text{-}input\ t = x \wedge t\text{-}output\ t = y)$
shows $(Inl\ (s1,s2),\ x,\ y,\ Inr\ q1) \in distinguishing\text{-}transitions\text{-}left\text{-}alt\ M\ q1\ q2$
⟨*proof*⟩

lemma *distinguishing-transitions-left-right-intro* :
assumes $(s1,s2) \in states\ (Product\text{-}FSM.product\ (FSM.from\text{-}FSM\ M\ q1)\ (FSM.from\text{-}FSM\ M\ q2))$
and $\neg(\exists\, t \in transitions\ M.\ t\text{-}source\ t = s1 \wedge t\text{-}input\ t = x \wedge t\text{-}output\ t = y)$
and $(\exists\, t \in transitions\ M.\ t\text{-}source\ t = s2 \wedge t\text{-}input\ t = x \wedge t\text{-}output\ t = y)$
shows $(Inl\ (s1,s2),\ x,\ y,\ Inr\ q2) \in distinguishing\text{-}transitions\text{-}right\text{-}alt\ M\ q1\ q2$
⟨*proof*⟩

lemma *canonical-separator-io-from-prefix-left* :
assumes *io* @ [*io1*] ∈ *LS M q1*
and *io* ∈ *LS M q2*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *observable M*
and *q1* ≠ *q2*
shows *io* @ [*io1*] ∈ *L* (*canonical-separator M q1 q2*)
⟨*proof*⟩

lemma *canonical-separator-path-targets-language* :
assumes *path* (*canonical-separator M q1 q2*) (*initial* (*canonical-separator M q1 q2*)) *p*
and *observable M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *q1* ≠ *q2*
shows *isl* (*target* (*initial* (*canonical-separator M q1 q2*)) *p*) ⟹ *p-io p* ∈ *LS M q1* ∩ *LS M q2*
and (*target* (*initial* (*canonical-separator M q1 q2*)) *p*) = *Inr q1* ⟹ *p-io p* ∈ *LS M q1* − *LS M q2* ∧ *p-io* (*butlast p*) ∈ *LS M q1* ∩ *LS M q2*
and (*target* (*initial* (*canonical-separator M q1 q2*)) *p*) = *Inr q2* ⟹ *p-io p* ∈ *LS M q2* − *LS M q1* ∧ *p-io* (*butlast p*) ∈ *LS M q1* ∩ *LS M q2*
and *p-io p* ∈ *LS M q1* ∩ *LS M q2* ⟹ *isl* (*target* (*initial* (*canonical-separator M q1 q2*)) *p*)
and *p-io p* ∈ *LS M q1* − *LS M q2* ⟹ *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inr q1*
and *p-io p* ∈ *LS M q2* − *LS M q1* ⟹ *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inr q2*
⟨*proof*⟩

lemma *canonical-separator-language-target* :
assumes *io* ∈ *L* (*canonical-separator M q1 q2*)
and *observable M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *q1* ≠ *q2*
shows *io* ∈ *LS M q1* − *LS M q2* ⟹ *io-targets* (*canonical-separator M q1 q2*) *io* (*initial* (*canonical-separator M q1 q2*)) = {*Inr q1*}
and *io* ∈ *LS M q2* − *LS M q1* ⟹ *io-targets* (*canonical-separator M q1 q2*) *io* (*initial* (*canonical-separator M q1 q2*)) = {*Inr q2*}
⟨*proof*⟩

lemma *canonical-separator-language-intersection* :
assumes *io* ∈ *LS M q1*
and *io* ∈ *LS M q2*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *io* ∈ *L* (*canonical-separator M q1 q2*) (**is** *io* ∈ *L ?C*)
⟨*proof*⟩

lemma *canonical-separator-deadlock* :
assumes *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *deadlock-state* (*canonical-separator M q1 q2*) (*Inr q1*)
and *deadlock-state* (*canonical-separator M q1 q2*) (*Inr q2*)
⟨*proof*⟩

lemma *canonical-separator-isl-deadlock* :
assumes *Inl* (*q1′*,*q2′*) ∈ *states* (*canonical-separator M q1 q2*)
and *x* ∈ *inputs M*
and *completely-specified M*
and ¬(∃ *t* ∈ *transitions* (*canonical-separator M q1 q2*) . *t-source t* = *Inl* (*q1′*,*q2′*) ∧ *t-input t* = *x* ∧ *isl* (*t-target t*))
and *q1* ∈ *states M*
and *q2* ∈ *states M*
obtains *y1 y2* **where** (*Inl* (*q1′*,*q2′*),*x*,*y1*,*Inr q1*) ∈ *transitions* (*canonical-separator M q1 q2*)
(*Inl* (*q1′*,*q2′*),*x*,*y2*,*Inr q2*) ∈ *transitions* (*canonical-separator M q1 q2*)
⟨*proof*⟩

lemma *canonical-separator-deadlocks* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows *deadlock-state* (*canonical-separator M q1 q2*) (*Inr q1*)
and *deadlock-state* (*canonical-separator M q1 q2*) (*Inr q2*)
⟨*proof*⟩

lemma *state-separator-from-canonical-separator-language-target* :
assumes *is-state-separator-from-canonical-separator* (*canonical-separator M q1 q2*) *q1 q2 A*
and *io* ∈ *L A*
and *observable M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *q1* ≠ *q2*
shows *io* ∈ *LS M q1* − *LS M q2* ⟹ *io-targets A io* (*initial A*) = {*Inr q1*}
and *io* ∈ *LS M q2* − *LS M q1* ⟹ *io-targets A io* (*initial A*) = {*Inr q2*}

and *io* ∈ *LS M q1* ∩ *LS M q2* ⟹ *io-targets A io* (*initial A*) ∩ {*Inr q1*, *Inr q2*} = {}
⟨*proof*⟩

lemma *state-separator-language-intersections-nonempty* :
assumes *is-state-separator-from-canonical-separator* (*canonical-separator M q1 q2*) *q1 q2 A*
and *observable M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *q1* ≠ *q2*
shows ∃ *io* . *io* ∈ (*L A* ∩ *LS M q1*) − *LS M q2* **and** ∃ *io* . *io* ∈ (*L A* ∩ *LS M q2*) − *LS M q1*
⟨*proof*⟩

lemma *state-separator-language-inclusion* :
assumes *is-state-separator-from-canonical-separator* (*canonical-separator M q1 q2*) *q1 q2 A*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *L A* ⊆ *LS M q1* ∪ *LS M q2*
⟨*proof*⟩

lemma *state-separator-from-canonical-separator-targets-left-inclusion* :
assumes *observable T*
and *observable M*
and *t1* ∈ *states T*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *is-state-separator-from-canonical-separator* (*canonical-separator M q1 q2*) *q1 q2 A*
and (*inputs T*) = (*inputs M*)
and *path A* (*initial A*) *p*
and *p-io p* ∈ *LS M q1*
and *q1* ≠ *q2*
shows *target* (*initial A*) *p* ≠ *Inr q2*
and *target* (*initial A*) *p* = *Inr q1* ∨ *isl* (*target* (*initial A*) *p*)
⟨*proof*⟩

lemma *state-separator-from-canonical-separator-targets-right-inclusion* :
assumes *observable T*
and *observable M*
and *t1* ∈ *states T*
and *q1* ∈ *states M*
and *q2* ∈ *states M*

and *is-state-separator-from-canonical-separator* (*canonical-separator M q1 q2*) *q1 q2 A*
and (*inputs T*) = (*inputs M*)
and *path A* (*initial A*) *p*
and *p-io p* ∈ *LS M q2*
and *q1* ≠ *q2*
shows *target* (*initial A*) *p* ≠ *Inr q1*
and *target* (*initial A*) *p* = *Inr q2* ∨ *isl* (*target* (*initial A*) *p*)
⟨*proof*⟩

34.2 Calculating State Separators

34.2.1 Sufficient Condition to Induce a State Separator

definition *state-separator-from-input-choices* :: ('*a*,'*b*,'*c*) *fsm* ⇒ (('*a* × '*a*) + '*a*,'*b*,'*c*) *fsm* ⇒ '*a* ⇒ '*a* ⇒ ((('*a* × '*a*) + '*a*) × '*b*) *list* ⇒ (('*a* × '*a*) + '*a*, '*b*, '*c*) *fsm* **where**
state-separator-from-input-choices M CSep q1 q2 cs =
(*let css* = *set cs*;
cssQ = (*set* (*map fst cs*)) ∪ {*Inr q1*, *Inr q2*};
S0 = *filter-states CSep* (λ *q* . *q* ∈ *cssQ*);
S1 = *filter-transitions S0* (λ *t* . (*t-source t*, *t-input t*) ∈ *css*)
in S1)

lemma *state-separator-from-input-choices-simps* :
assumes *q1* ∈ *states M*
and *q2* ∈ *states M*
and ⋀ *qq x* . (*qq*,*x*) ∈ *set cs* ⟹ *qq* ∈ *states* (*canonical-separator M q1 q2*) ∧ *x* ∈ *inputs M*
and *Inl* (*q1*,*q2*) ∈ *set* (*map fst cs*)
and ⋀ *qq* . *qq* ∈ *set* (*map fst cs*) ⟹ ∃ *q1' q2'* . *qq* = *Inl* (*q1'*,*q2'*)
shows
initial (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) = *Inl* (*q1*,*q2*)
states (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) = (*set* (*map fst cs*)) ∪ {*Inr q1*, *Inr q2*}
inputs (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) = *inputs M*
outputs (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) = *outputs M*
transitions (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) =
{*t* ∈ (*transitions* (*canonical-separator M q1 q2*)) . ∃ *q1' q2' x* . (*Inl* (*q1'*,*q2'*),*x*) ∈ *set cs* ∧ *t-source t* = *Inl* (*q1'*,*q2'*) ∧ *t-input t* = *x* ∧ *t-target t* ∈ (*set* (*map fst cs*)) ∪ {*Inr q1*, *Inr q2*}}
⟨*proof*⟩

lemma *state-separator-from-input-choices-submachine* :
assumes *q1* ∈ *states M*
and *q2* ∈ *states M*
and ⋀ *qq x* . (*qq*,*x*) ∈ *set cs* ⟹ *qq* ∈ *states* (*canonical-separator M q1 q2*) ∧ *x* ∈ *inputs M*
and *Inl* (*q1*,*q2*) ∈ *set* (*map fst cs*)
and ⋀ *qq* . *qq* ∈ *set* (*map fst cs*) ⟹ ∃ *q1' q2'* . *qq* = *Inl* (*q1'*,*q2'*)
shows *is-submachine* (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) (*canonical-separator M q1 q2*)
⟨*proof*⟩

lemma *state-separator-from-input-choices-single-input* :
assumes *distinct* (*map fst cs*)
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and ⋀ *qq x* . (*qq*,*x*) ∈ *set cs* ⟹ *qq* ∈ *states* (*canonical-separator M q1 q2*) ∧ *x* ∈ *inputs M*
and *Inl* (*q1*,*q2*) ∈ *set* (*map fst cs*)
and ⋀ *qq* . *qq* ∈ *set* (*map fst cs*) ⟹ ∃ *q1' q2'* . *qq* = *Inl* (*q1'*,*q2'*)
shows *single-input* (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*)
⟨*proof*⟩

lemma *state-separator-from-input-choices-transition-list* :
assumes *q1* ∈ *states M*
and *q2* ∈ *states M*
and ⋀ *qq x* . (*qq*,*x*) ∈ *set cs* ⟹ *qq* ∈ *states* (*canonical-separator M q1 q2*) ∧ *x* ∈ *inputs M*
and *Inl* (*q1*,*q2*) ∈ *set* (*map fst cs*)
and ⋀ *qq* . *qq* ∈ *set* (*map fst cs*) ⟹ ∃ *q1' q2'* . *qq* = *Inl* (*q1'*,*q2'*)
and *t* ∈ *transitions* (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*)
shows (*t-source t*, *t-input t*) ∈ *set cs*
⟨*proof*⟩

lemma *state-separator-from-input-choices-transition-target* :
assumes *t* ∈ *transitions* (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*)
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and ⋀ *qq x* . (*qq*,*x*) ∈ *set cs* ⟹ *qq* ∈ *states* (*canonical-separator M q1 q2*) ∧ *x* ∈ *inputs M*
and *Inl* (*q1*,*q2*) ∈ *set* (*map fst cs*)
and ⋀ *qq* . *qq* ∈ *set* (*map fst cs*) ⟹ ∃ *q1' q2'* . *qq* = *Inl* (*q1'*,*q2'*)
shows *t* ∈ *transitions* (*canonical-separator M q1 q2*) ∨ *t-target t* ∈ {*Inr q1*, *Inr q2*}

⟨proof⟩

lemma *state-separator-from-input-choices-acyclic-paths′* :
assumes *distinct* (*map fst cs*)
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and ⋀ *qq x* . (*qq*,*x*) ∈ *set cs* ⟹ *qq* ∈ *states* (*canonical-separator M q1 q2*) ∧ *x* ∈ *inputs M*
and *Inl* (*q1*,*q2*) ∈ *set* (*map fst cs*)
and ⋀ *qq* . *qq* ∈ *set* (*map fst cs*) ⟹ ∃ *q1′ q2′* . *qq* = *Inl* (*q1′*,*q2′*)
and ⋀ *i t* . *i* < *length cs*
⟹ *t* ∈ *transitions* (*canonical-separator M q1 q2*)
⟹ *t-source t* = (*fst* (*cs* ! *i*))
⟹ *t-input t* = *snd* (*cs* ! *i*)
⟹ *t-target t* ∈ ((*set* (*map fst* (*take i cs*))) ∪ {*Inr q1*, *Inr q2*})
and *path* (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) *q′ p*
and *target q′ p* = *q′*
and *p* ≠ []
shows *False*
⟨proof⟩

lemma *state-separator-from-input-choices-acyclic-paths* :
assumes *distinct* (*map fst cs*)
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and ⋀ *qq x* . (*qq*,*x*) ∈ *set cs* ⟹ *qq* ∈ *states* (*canonical-separator M q1 q2*) ∧ *x* ∈ *inputs M*
and *Inl* (*q1*,*q2*) ∈ *set* (*map fst cs*)
and ⋀ *qq* . *qq* ∈ *set* (*map fst cs*) ⟹ ∃ *q1′ q2′* . *qq* = *Inl* (*q1′*,*q2′*)
and ⋀ *i t* . *i* < *length cs*
⟹ *t* ∈ *transitions* (*canonical-separator M q1 q2*)
⟹ *t-source t* = (*fst* (*cs* ! *i*))
⟹ *t-input t* = *snd* (*cs* ! *i*)
⟹ *t-target t* ∈ ((*set* (*map fst* (*take i cs*))) ∪ {*Inr q1*, *Inr q2*})
and *path* (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) *q′ p*
shows *distinct* (*visited-states q′ p*)
⟨proof⟩

lemma *state-separator-from-input-choices-acyclic* :
assumes *distinct* (*map fst cs*)
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and ⋀ *qq x* . (*qq*,*x*) ∈ *set cs* ⟹ *qq* ∈ *states* (*canonical-separator M q1 q2*) ∧ *x* ∈ *inputs M*

and *Inl* (*q1*,*q2*) ∈ *set* (*map fst cs*)
and ⋀ *qq* . *qq* ∈ *set* (*map fst cs*) ⟹ ∃ *q1′ q2′* . *qq* = *Inl* (*q1′*,*q2′*)
and ⋀ *i t* . *i* < *length cs*
⟹ *t* ∈ *transitions* (*canonical-separator M q1 q2*)
⟹ *t-source t* = (*fst* (*cs* ! *i*))
⟹ *t-input t* = *snd* (*cs* ! *i*)
⟹ *t-target t* ∈ ((*set* (*map fst* (*take i cs*))) ∪ {*Inr q1*, *Inr q2*})
shows *acyclic* (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*)
⟨*proof*⟩

lemma *state-separator-from-input-choices-target* :
assumes ⋀ *i t* . *i* < *length cs*
⟹ *t* ∈ *transitions* (*canonical-separator M q1 q2*)
⟹ *t-source t* = (*fst* (*cs* ! *i*))
⟹ *t-input t* = *snd* (*cs* ! *i*)
⟹ *t-target t* ∈ ((*set* (*map fst* (*take i cs*))) ∪ {*Inr q1*, *Inr q2*})
and *t* ∈ *FSM.transitions* (*canonical-separator M q1 q2*)
and ∃ *q1′ q2′ x* . (*Inl* (*q1′*, *q2′*), *x*)∈*set cs* ∧ *t-source t* = *Inl* (*q1′*, *q2′*) ∧ *t-input t* = *x*
shows *t-target t* ∈ *set* (*map fst cs*) ∪ {*Inr q1*, *Inr q2*}
⟨*proof*⟩

lemma *state-separator-from-input-choices-transitions-alt-def* :
assumes *q1* ∈ *states M*
and *q2* ∈ *states M*
and ⋀ *qq x* . (*qq*,*x*) ∈ *set cs* ⟹ *qq* ∈ *states* (*canonical-separator M q1 q2*) ∧ *x* ∈ *inputs M*
and *Inl* (*q1*,*q2*) ∈ *set* (*map fst cs*)
and ⋀ *qq* . *qq* ∈ *set* (*map fst cs*) ⟹ ∃ *q1′ q2′* . *qq* = *Inl* (*q1′*,*q2′*)
and ⋀ *i t* . *i* < *length cs*
⟹ *t* ∈ *transitions* (*canonical-separator M q1 q2*)
⟹ *t-source t* = (*fst* (*cs* ! *i*))
⟹ *t-input t* = *snd* (*cs* ! *i*)
⟹ *t-target t* ∈ ((*set* (*map fst* (*take i cs*))) ∪ {*Inr q1*, *Inr q2*})
shows *transitions* (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) =
{*t* ∈ (*transitions* (*canonical-separator M q1 q2*)) . ∃ *q1′ q2′ x* . (*Inl* (*q1′*,*q2′*),*x*) ∈ *set cs* ∧ *t-source t* = *Inl* (*q1′*,*q2′*) ∧ *t-input t* = *x*}
⟨*proof*⟩

lemma *state-separator-from-input-choices-deadlock* :
assumes *distinct* (*map fst cs*)
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and ⋀ *qq x* . (*qq*,*x*) ∈ *set cs* ⟹ *qq* ∈ *states* (*canonical-separator M q1 q2*)

∧ *x* ∈ *inputs M*
  and *Inl* (*q1*,*q2*) ∈ *set* (*map fst cs*)
  and ⋀ *qq* . *qq* ∈ *set* (*map fst cs*) ⟹ ∃ *q1′ q2′* . *qq* = *Inl* (*q1′*,*q2′*)
  and ⋀ *i t* . *i* < *length cs*
    ⟹ *t* ∈ *transitions* (*canonical-separator M q1 q2*)
    ⟹ *t-source t* = (*fst* (*cs* ! *i*))
    ⟹ *t-input t* = *snd* (*cs* ! *i*)
    ⟹ *t-target t* ∈ ((*set* (*map fst* (*take i cs*))) ∪ {*Inr q1*, *Inr q2*})

 shows ⋀ *qq* . *qq* ∈ *states* (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) ⟹ *deadlock-state* (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) *qq* ⟹ *qq* ∈ {*Inr q1*, *Inr q2*} ∨ (∃ *q1′ q2′ x* . *qq* = *Inl* (*q1′*,*q2′*) ∧ *x* ∈ *inputs M* ∧ (*h-out M* (*q1′*,*x*) = {} ∧ *h-out M* (*q2′*,*x*) = {}))
⟨*proof*⟩

lemma *state-separator-from-input-choices-retains-io* :
 assumes *distinct* (*map fst cs*)
  and *q1* ∈ *states M*
  and *q2* ∈ *states M*
  and ⋀ *qq x* . (*qq*,*x*) ∈ *set cs* ⟹ *qq* ∈ *states* (*canonical-separator M q1 q2*) ∧ *x* ∈ *inputs M*
  and *Inl* (*q1*,*q2*) ∈ *set* (*map fst cs*)
  and ⋀ *qq* . *qq* ∈ *set* (*map fst cs*) ⟹ ∃ *q1′ q2′* . *qq* = *Inl* (*q1′*,*q2′*)
  and ⋀ *i t* . *i* < *length cs*
    ⟹ *t* ∈ *transitions* (*canonical-separator M q1 q2*)
    ⟹ *t-source t* = (*fst* (*cs* ! *i*))
    ⟹ *t-input t* = *snd* (*cs* ! *i*)
    ⟹ *t-target t* ∈ ((*set* (*map fst* (*take i cs*))) ∪ {*Inr q1*, *Inr q2*})
 shows *retains-outputs-for-states-and-inputs* (*canonical-separator M q1 q2*) (*state-separator-from-input-choic* *M* (*canonical-separator M q1 q2*) *q1 q2 cs*)
 ⟨*proof*⟩

lemma *state-separator-from-input-choices-is-state-separator* :
 assumes *distinct* (*map fst cs*)
  and *q1* ∈ *states M*
  and *q2* ∈ *states M*
  and ⋀ *qq x* . (*qq*,*x*) ∈ *set cs* ⟹ *qq* ∈ *states* (*canonical-separator M q1 q2*) ∧ *x* ∈ *inputs M*
  and *Inl* (*q1*,*q2*) ∈ *set* (*map fst cs*)
  and ⋀ *qq* . *qq* ∈ *set* (*map fst cs*) ⟹ ∃ *q1′ q2′* . *qq* = *Inl* (*q1′*,*q2′*)
  and ⋀ *i t* . *i* < *length cs*
    ⟹ *t* ∈ *transitions* (*canonical-separator M q1 q2*)
    ⟹ *t-source t* = (*fst* (*cs* ! *i*))
    ⟹ *t-input t* = *snd* (*cs* ! *i*)
    ⟹ *t-target t* ∈ ((*set* (*map fst* (*take i cs*))) ∪ {*Inr q1*, *Inr q2*})
  and *completely-specified M*
 shows *is-state-separator-from-canonical-separator*

(*canonical-separator M q1 q2*)
q1
q2
(*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*)
⟨*proof*⟩

34.2.2 Calculating a State Separator by Backwards Reachability Analysis

A state separator for states *q1* and *q2* can be calculated using backwards reachability analysis starting from the two deadlock states of their canonical separator until *Inl* (*q1*.*q2*) is reached or it is not possible to reach (*q1*,*q2*).

definition *s-states* :: (′*a*::*linorder*,′*b*::*linorder*,′*c*) *fsm* ⇒ ′*a* ⇒ ′*a* ⇒ (((′*a* × ′*a*) + ′*a*) × ′*b*) *list* **where**
s-states M q1 q2 = (*let C* = *canonical-separator M q1 q2*
in select-inputs (*h C*) (*initial C*) (*inputs-as-list C*) (*remove1* (*Inl* (*q1*,*q2*)) (*remove1* (*Inr q1*) (*remove1* (*Inr q2*) (*states-as-list C*)))) {*Inr q1*, *Inr q2*} [])

definition *state-separator-from-s-states* :: (′*a*::*linorder*,′*b*::*linorder*,′*c*) *fsm* ⇒ ′*a* ⇒ ′*a* ⇒ ((′*a* × ′*a*) + ′*a*, ′*b*, ′*c*) *fsm option*
where
state-separator-from-s-states M q1 q2 =
(*let cs* = *s-states M q1 q2*
in (*case length cs of*
0 ⇒ *None* |
- ⇒ *if fst* (*last cs*) = *Inl* (*q1*,*q2*)
then Some (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*)
else None))

lemma *state-separator-from-s-states-code*[*code*] :
state-separator-from-s-states M q1 q2 =
(*let C* = *canonical-separator M q1 q2*;
cs = *select-inputs* (*h C*) (*initial C*) (*inputs-as-list C*) (*remove1* (*Inl* (*q1*,*q2*)) (*remove1* (*Inr q1*) (*remove1* (*Inr q2*) (*states-as-list C*)))) {*Inr q1*, *Inr q2*} []
in (*case length cs of*
0 ⇒ *None* |
- ⇒ *if fst* (*last cs*) = *Inl* (*q1*,*q2*)
then Some (*state-separator-from-input-choices M C q1 q2 cs*)
else None))
⟨*proof*⟩

lemma *s-states-properties* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*

shows *distinct* (*map fst* (*s-states M q1 q2*))
and ⋀ *qq x* . (*qq*,*x*) ∈ *set* (*s-states M q1 q2*) ⟹ *qq* ∈ *states* (*canonical-separator M q1 q2*) ∧ *x* ∈ *inputs M*
and ⋀ *qq* . *qq* ∈ *set* (*map fst* (*s-states M q1 q2*)) ⟹ ∃ *q1′ q2′* . *qq* = *Inl* (*q1′*,*q2′*)
and ⋀ *i t* . *i* < *length* (*s-states M q1 q2*)
⟹ *t* ∈ *transitions* (*canonical-separator M q1 q2*)
⟹ *t-source t* = (*fst* ((*s-states M q1 q2*) ! *i*))
⟹ *t-input t* = *snd* ((*s-states M q1 q2*) ! *i*)
⟹ *t-target t* ∈ ((*set* (*map fst* (*take i* (*s-states M q1 q2*)))) ∪ {*Inr q1*, *Inr q2*})
⟨*proof*⟩

lemma *state-separator-from-s-states-soundness* :
assumes *state-separator-from-s-states M q1 q2* = *Some A*
and *q1* ∈ *states M* **and** *q2* ∈ *states M* **and** *completely-specified M*
shows *is-state-separator-from-canonical-separator* (*canonical-separator M q1 q2*) *q1 q2 A*
⟨*proof*⟩

lemma *state-separator-from-s-states-exhaustiveness* :
assumes ∃ *S* . *is-state-separator-from-canonical-separator* (*canonical-separator M q1 q2*) *q1 q2 S*
and *q1* ∈ *states M* **and** *q2* ∈ *states M* **and** *completely-specified M* **and** *observable M*
shows *state-separator-from-s-states M q1 q2* ≠ *None*
⟨*proof*⟩

34.3 Generalizing State Separators

State separators can be defined without reverence to the canonical separator:

definition *is-separator* :: (′*a*,′*b*,′*c*) *fsm* ⇒ ′*a* ⇒ ′*a* ⇒ (′*d*,′*b*,′*c*) *fsm* ⇒ ′*d* ⇒ ′*d* ⇒ *bool* **where**
is-separator M q1 q2 A t1 t2 =
(*single-input A*
∧ *acyclic A*
∧ *observable A*
∧ *deadlock-state A t1*
∧ *deadlock-state A t2*
∧ *t1* ∈ *reachable-states A*
∧ *t2* ∈ *reachable-states A*
∧ (∀ *t* ∈ *reachable-states A* . (*t* ≠ *t1* ∧ *t* ≠ *t2*) ⟶ ¬ *deadlock-state A t*)
∧ (∀ *io* ∈ *L A* . (∀ *x yq yt* . (*io*@[(*x*,*yq*)] ∈ *LS M q1* ∧ *io*@[(*x*,*yt*)] ∈ *L A*) ⟶ (*io*@[(*x*,*yq*)] ∈ *L A*))
∧ (∀ *x yq2 yt* . (*io*@[(*x*,*yq2*)] ∈ *LS M q2* ∧ *io*@[(*x*,*yt*)] ∈ *L A*) ⟶ (*io*@[(*x*,*yq2*)] ∈ *L A*)))
∧ (∀ *p* . (*path A* (*initial A*) *p* ∧ *target* (*initial A*) *p* = *t1*) ⟶ *p-io p* ∈ *LS M*

q1 − *LS M q2*)
∧ (∀ *p* . (*path A* (*initial A*) *p* ∧ *target* (*initial A*) *p* = *t2*) ⟶ *p-io p* ∈ *LS M q2* − *LS M q1*)
∧ (∀ *p* . (*path A* (*initial A*) *p* ∧ *target* (*initial A*) *p* ≠ *t1* ∧ *target* (*initial A*) *p* ≠ *t2*) ⟶ *p-io p* ∈ *LS M q1* ∩ *LS M q2*)
∧ *q1* ≠ *q2*
∧ *t1* ≠ *t2*
∧ (*inputs A*) ⊆ (*inputs M*))

lemma *is-separator-simps* :
assumes *is-separator M q1 q2 A t1 t2*
shows *single-input A*
and *acyclic A*
and *observable A*
and *deadlock-state A t1*
and *deadlock-state A t2*
and *t1* ∈ *reachable-states A*
and *t2* ∈ *reachable-states A*
and ⋀ *t* . *t* ∈ *reachable-states A* ⟹ *t* ≠ *t1* ⟹ *t* ≠ *t2* ⟹ ¬ *deadlock-state A t*
and ⋀ *io x yq yt* . *io*@[(*x*,*yq*)] ∈ *LS M q1* ⟹ *io*@[(*x*,*yt*)] ∈ *L A* ⟹ (*io*@[(*x*,*yq*)] ∈ *L A*)
and ⋀ *io x yq yt* . *io*@[(*x*,*yq*)] ∈ *LS M q2* ⟹ *io*@[(*x*,*yt*)] ∈ *L A* ⟹ (*io*@[(*x*,*yq*)] ∈ *L A*)
and ⋀ *p* . *path A* (*initial A*) *p* ⟹ *target* (*initial A*) *p* = *t1* ⟹ *p-io p* ∈ *LS M q1* − *LS M q2*
and ⋀ *p* . *path A* (*initial A*) *p* ⟹ *target* (*initial A*) *p* = *t2* ⟹ *p-io p* ∈ *LS M q2* − *LS M q1*
and ⋀ *p* . *path A* (*initial A*) *p* ⟹ *target* (*initial A*) *p* ≠ *t1* ⟹ *target* (*initial A*) *p* ≠ *t2* ⟹ *p-io p* ∈ *LS M q1* ∩ *LS M q2*
and *q1* ≠ *q2*
and *t1* ≠ *t2*
and (*inputs A*) ⊆ (*inputs M*)
⟨*proof*⟩

lemma *separator-initial* :
assumes *is-separator M q1 q2 A t1 t2*
shows *initial A* ≠ *t1*
and *initial A* ≠ *t2*
⟨*proof*⟩

lemma *separator-path-targets* :
assumes *is-separator M q1 q2 A t1 t2*
and *path A* (*initial A*) *p*
shows *p-io p* ∈ *LS M q1* − *LS M q2* ⟹ *target* (*initial A*) *p* = *t1*
and *p-io p* ∈ *LS M q2* − *LS M q1* ⟹ *target* (*initial A*) *p* = *t2*
and *p-io p* ∈ *LS M q1* ∩ *LS M q2* ⟹ (*target* (*initial A*) *p* ≠ *t1* ∧ *target* (*initial*

A) *p* ≠ *t2*)
and *p-io p* ∈ *LS M q1* ∪ *LS M q2*
⟨*proof*⟩

lemma *separator-language* :
assumes *is-separator M q1 q2 A t1 t2*
and *io* ∈ *L A*
shows *io* ∈ *LS M q1* − *LS M q2* ⟹ *io-targets A io* (*initial A*) = {*t1*}
and *io* ∈ *LS M q2* − *LS M q1* ⟹ *io-targets A io* (*initial A*) = {*t2*}
and *io* ∈ *LS M q1* ∩ *LS M q2* ⟹ *io-targets A io* (*initial A*) ∩ {*t1*,*t2*} = {}
and *io* ∈ *LS M q1* ∪ *LS M q2*
⟨*proof*⟩

lemma *is-separator-sym* :
is-separator M q1 q2 A t1 t2 ⟹ *is-separator M q2 q1 A t2 t1*
⟨*proof*⟩

lemma *state-separator-from-canonical-separator-is-separator* :
assumes *is-state-separator-from-canonical-separator* (*canonical-separator M q1 q2*) *q1 q2 A*
and *observable M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *q1* ≠ *q2*
shows *is-separator M q1 q2 A* (*Inr q1*) (*Inr q2*)
⟨*proof*⟩

lemma *is-separator-separated-state-is-state* :
assumes *is-separator M q1 q2 A t1 t2*
shows *q1* ∈ *states M* **and** *q2* ∈ *states M*
⟨*proof*⟩

end

35 Adaptive Test Cases

An ATC is a single input, acyclic, observable FSM, which is equivalent to a tree whose non-leaf states are labeled with inputs and whose edges are labeled with outputs.

theory *Adaptive-Test-Case*
imports *State-Separator*
begin

definition *is-ATC* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ *bool* **where**
is-ATC M = (*single-input M* $\wedge$ *acyclic M* $\wedge$ *observable M*)

lemma *is-ATC-from* :
assumes $t \in$ *transitions A*
and *t-source t* $\in$ *reachable-states A*
and *is-ATC A*
shows *is-ATC* (*from-FSM A* (*t-target t*))
⟨*proof*⟩

35.1 Applying Adaptive Test Cases

fun *pass-ATC'* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $('d,'b,'c)$ *fsm* $\Rightarrow$ *'d set* $\Rightarrow$ *nat* $\Rightarrow$ *bool* **where**
pass-ATC' M A fail-states 0 = ($\neg$ (*initial A* $\in$ *fail-states*)) |
pass-ATC' M A fail-states (*Suc k*) = (($\neg$ (*initial A* $\in$ *fail-states*)) $\wedge$
($\forall$ *x* $\in$ *inputs A* . *h A* (*initial A*,*x*) $\neq$ {} $\longrightarrow$ ($\forall$ (*yM*,*qM*) $\in$ *h M* (*initial M*,*x*) . $\exists$ (*yA*,*qA*) $\in$ *h A* (*initial A*,*x*) . *yM* = *yA* $\wedge$ *pass-ATC'* (*from-FSM M qM*) (*from-FSM A qA*) *fail-states k*)))

fun *pass-ATC* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $('d,'b,'c)$ *fsm* $\Rightarrow$ *'d set* $\Rightarrow$ *bool* **where**
pass-ATC M A fail-states = *pass-ATC' M A fail-states* (*size A*)

lemma *pass-ATC'-initial* :
assumes *pass-ATC' M A FS k*
shows *initial A* $\notin$ *FS*
⟨*proof*⟩

lemma *pass-ATC'-io* :
assumes *pass-ATC' M A FS k*
and *is-ATC A*
and *observable M*
and (*inputs A*) $\subseteq$ (*inputs M*)
and *io*@[*ioA*] $\in$ *L A*
and *io*@[*ioM*] $\in$ *L M*
and *fst ioA* = *fst ioM*
and *length* (*io*@[*ioA*]) $\leq$ *k*
shows *io*@[*ioM*] $\in$ *L A*
and *io-targets A* (*io*@[*ioM*]) (*initial A*) $\cap$ *FS* = {}
⟨*proof*⟩

lemma *pass-ATC-io* :
assumes *pass-ATC M A FS*
and *is-ATC A*
and *observable M*

and $(inputs\ A) \subseteq (inputs\ M)$
and $io@[ioA] \in L\ A$
and $io@[ioM] \in L\ M$
and $fst\ ioA = fst\ ioM$
shows $io@[ioM] \in L\ A$
and $io\text{-}targets\ A\ (io@[ioM])\ (initial\ A) \cap FS = \{\}$
⟨*proof*⟩

lemma *pass-ATC-io-explicit-io-tuple* :
assumes $pass\text{-}ATC\ M\ A\ FS$
and $is\text{-}ATC\ A$
and $observable\ M$
and $(inputs\ A) \subseteq (inputs\ M)$
and $io@[(x,y)] \in L\ A$
and $io@[(x,y')] \in L\ M$
shows $io@[(x,y')] \in L\ A$
and $io\text{-}targets\ A\ (io@[(x,y')])\ (initial\ A) \cap FS = \{\}$
⟨*proof*⟩

lemma *pass-ATC-io-fail-fixed-io* :
assumes $is\text{-}ATC\ A$
and $observable\ M$
and $(inputs\ A) \subseteq (inputs\ M)$
and $io@[ioA] \in L\ A$
and $io@[ioM] \in L\ M$
and $fst\ ioA = fst\ ioM$
and $io@[ioM] \notin L\ A \vee io\text{-}targets\ A\ (io@[ioM])\ (initial\ A) \cap FS \neq \{\}$
shows $\neg pass\text{-}ATC\ M\ A\ FS$
⟨*proof*⟩

lemma *pass-ATC′-io-fail* :
assumes $\neg pass\text{-}ATC'\ M\ A\ FS\ k$
and $is\text{-}ATC\ A$
and $observable\ M$
and $(inputs\ A) \subseteq (inputs\ M)$
shows $initial\ A \in FS \vee (\exists\ io\ ioA\ ioM\ .\ io@[ioA] \in L\ A$
$\wedge\ io@[ioM] \in L\ M$
$\wedge\ fst\ ioA = fst\ ioM$
$\wedge\ (io@[ioM] \notin L\ A \vee io\text{-}targets\ A\ (io@[ioM])\ (initial\ A) \cap$
$FS \neq \{\}))$
⟨*proof*⟩

lemma *pass-ATC-io-fail* :
assumes $\neg pass\text{-}ATC\ M\ A\ FS$
and $is\text{-}ATC\ A$

and *observable M*
and (*inputs A*) ⊆ (*inputs M*)
shows *initial A* ∈ *FS* ∨ (∃ *io ioA ioM* . *io*@[*ioA*] ∈ *L A*
∧ *io*@[*ioM*] ∈ *L M*
∧ *fst ioA* = *fst ioM*
∧ (*io*@[*ioM*] ∉ *L A* ∨ *io-targets A* (*io*@[*ioM*]) (*initial A*) ∩ *FS* ≠ {}))
⟨*proof*⟩

lemma *pass-ATC-fail* :
assumes *is-ATC A*
and *observable M*
and (*inputs A*) ⊆ (*inputs M*)
and *io*@[(*x*,*y*)] ∈ *L A*
and *io*@[(*x*,*y'*)] ∈ *L M*
and *io*@[(*x*,*y'*)] ∉ *L A*
shows ¬ *pass-ATC M A FS*
⟨*proof*⟩

lemma *pass-ATC-reduction* :
assumes *L M2* ⊆ *L M1*
and *is-ATC A*
and *observable M1*
and *observable M2*
and (*inputs A*) ⊆ (*inputs M1*)
and (*inputs M2*) = (*inputs M1*)
and *pass-ATC M1 A FS*
shows *pass-ATC M2 A FS*
⟨*proof*⟩

lemma *pass-ATC-fail-no-reduction* :
assumes *is-ATC A*
and *observable T*
and *observable M*
and (*inputs A*) ⊆ (*inputs M*)
and (*inputs T*) = (*inputs M*)
and *pass-ATC M A FS*
and ¬*pass-ATC T A FS*
shows ¬ (*L T* ⊆ *L M*)
⟨*proof*⟩

35.2 State Separators as Adaptive Test Cases

fun *pass-separator-ATC* :: (*'a*,*'b*,*'c*) *fsm* ⇒ (*'d*,*'b*,*'c*) *fsm* ⇒ *'a* ⇒ *'d* ⇒ *bool* **where**
pass-separator-ATC M S q1 t2 = *pass-ATC* (*from-FSM M q1*) *S* {*t2*}

lemma *separator-is-ATC* :
assumes *is-separator M q1 q2 A t1 t2*
and *observable M*
and *q1* ∈ *states M*
shows *is-ATC A*
⟨*proof*⟩

lemma *pass-separator-ATC-from-separator-left* :
assumes *observable M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *is-separator M q1 q2 A t1 t2*
shows *pass-separator-ATC M A q1 t2*
⟨*proof*⟩

lemma *pass-separator-ATC-from-separator-right* :
assumes *observable M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *is-separator M q1 q2 A t1 t2*
shows *pass-separator-ATC M A q2 t1*
⟨*proof*⟩

lemma *pass-separator-ATC-path-left* :
assumes *pass-separator-ATC S A s1 t2*
and *observable S*
and *observable M*
and *s1* ∈ *states S*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *is-separator M q1 q2 A t1 t2*
and (*inputs S*) = (*inputs M*)
and *q1* ≠ *q2*
and *path A* (*initial A*) *pA*
and *path S s1 pS*
and *p-io pA* = *p-io pS*
shows *target* (*initial A*) *pA* ≠ *t2*
and ∃ *pM* . *path M q1 pM* ∧ *p-io pM* = *p-io pA*
⟨*proof*⟩

lemma *pass-separator-ATC-path-right* :
assumes *pass-separator-ATC S A s2 t1*
and *observable S*
and *observable M*

and *s2* ∈ *states S*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *is-separator M q1 q2 A t1 t2*
and (*inputs S*) = (*inputs M*)
and *q1* ≠ *q2*
and *path A* (*initial A*) *pA*
and *path S s2 pS*
and *p-io pA* = *p-io pS*
shows *target* (*initial A*) *pA* ≠ *t1*
and ∃ *pM* . *path M q2 pM* ∧ *p-io pM* = *p-io pA*
⟨*proof*⟩

lemma *pass-separator-ATC-fail-no-reduction* :
assumes *observable S*
and *observable M*
and *s1* ∈ *states S*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *is-separator M q1 q2 A t1 t2*
and (*inputs S*) = (*inputs M*)
and ¬*pass-separator-ATC S A s1 t2*
shows ¬ (*LS S s1* ⊆ *LS M q1*)
⟨*proof*⟩

lemma *pass-separator-ATC-pass-left* :
assumes *observable S*
and *observable M*
and *s1* ∈ *states S*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *is-separator M q1 q2 A t1 t2*
and (*inputs S*) = (*inputs M*)
and *path A* (*initial A*) *p*
and *p-io p* ∈ *LS S s1*
and *q1* ≠ *q2*
and *pass-separator-ATC S A s1 t2*
shows *target* (*initial A*) *p* ≠ *t2*
and *target* (*initial A*) *p* = *t1* ∨ (*target* (*initial A*) *p* ≠ *t1* ∧ *target* (*initial A*) *p* ≠ *t2*)
⟨*proof*⟩

lemma *pass-separator-ATC-pass-right* :
assumes *observable S*
and *observable M*
and *s2* ∈ *states S*

and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *is-separator M q1 q2 A t1 t2*
and (*inputs S*) = (*inputs M*)
and *path A* (*initial A*) *p*
and *p-io p* ∈ *LS S s2*
and *q1* ≠ *q2*
and *pass-separator-ATC S A s2 t1*
shows *target* (*initial A*) *p* ≠ *t1*
and *target* (*initial A*) *p* = *t2* ∨ (*target* (*initial A*) *p* ≠ *t2* ∧ *target* (*initial A*) *p* ≠ *t2*)
⟨*proof*⟩

lemma *pass-separator-ATC-completely-specified-left* :
assumes *observable S*
and *observable M*
and *s1* ∈ *states S*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *is-separator M q1 q2 A t1 t2*
and (*inputs S*) = (*inputs M*)
and *q1* ≠ *q2*
and *pass-separator-ATC S A s1 t2*
and *completely-specified S*
shows ∃ *p* . *path A* (*initial A*) *p* ∧ *p-io p* ∈ *LS S s1* ∧ *target* (*initial A*) *p* = *t1*
and ¬ (∃ *p* . *path A* (*initial A*) *p* ∧ *p-io p* ∈ *LS S s1* ∧ *target* (*initial A*) *p* = *t2*)
⟨*proof*⟩

lemma *pass-separator-ATC-completely-specified-right* :
assumes *observable S*
and *observable M*
and *s2* ∈ *states S*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *is-separator M q1 q2 A t1 t2*
and (*inputs S*) = (*inputs M*)
and *q1* ≠ *q2*
and *pass-separator-ATC S A s2 t1*
and *completely-specified S*
shows ∃ *p* . *path A* (*initial A*) *p* ∧ *p-io p* ∈ *LS S s2* ∧ *target* (*initial A*) *p* = *t2*
and ¬ (∃ *p* . *path A* (*initial A*) *p* ∧ *p-io p* ∈ *LS S s2* ∧ *target* (*initial A*) *p* = *t1*)
⟨*proof*⟩

lemma *pass-separator-ATC-reduction-distinction* :

assumes *observable M*
and *observable S*
and (*inputs S*) = (*inputs M*)
and *pass-separator-ATC S A s1 t2*
and *pass-separator-ATC S A s2 t1*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *q1* ≠ *q2*
and *s1* ∈ *states S*
and *s2* ∈ *states S*
and *is-separator M q1 q2 A t1 t2*
and *completely-specified S*
shows *s1* ≠ *s2*
⟨*proof*⟩

lemma *pass-separator-ATC-failure-left* :
assumes *observable M*
and *observable S*
and (*inputs S*) = (*inputs M*)
and *is-separator M q1 q2 A t1 t2*
and ¬ *pass-separator-ATC S A s1 t2*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *q1* ≠ *q2*
and *s1* ∈ *states S*
shows *LS S s1* − *LS M q1* ≠ {}
⟨*proof*⟩

lemma *pass-separator-ATC-failure-right* :
assumes *observable M*
and *observable S*
and (*inputs S*) = (*inputs M*)
and *is-separator M q1 q2 A t1 t2*
and ¬ *pass-separator-ATC S A s2 t1*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *q1* ≠ *q2*
and *s2* ∈ *states S*
shows *LS S s2* − *LS M q2* ≠ {}
⟨*proof*⟩

35.3 ATCs Represented as Sets of IO Sequences

fun *atc-to-io-set* :: (′*a*,′*b*,′*c*) *fsm* ⇒ (′*d*,′*b*,′*c*) *fsm* ⇒ (′*b* × ′*c*) *list set* **where**
atc-to-io-set M A = *L M* ∩ *L A*

lemma *atc-to-io-set-code* :
assumes *acyclic A*
shows *atc-to-io-set M A* = *acyclic-language-intersection M A*
⟨*proof*⟩

lemma *pass-io-set-from-pass-separator* :
assumes *is-separator M q1 q2 A t1 t2*
and *pass-separator-ATC S A s1 t2*
and *observable M*
and *observable S*
and *q1* ∈ *states M*
and *s1* ∈ *states S*
and (*inputs S*) = (*inputs M*)
shows *pass-io-set* (*from-FSM S s1*) (*atc-to-io-set* (*from-FSM M q1*) *A*)
⟨*proof*⟩

lemma *separator-language-last-left* :
assumes *is-separator M q1 q2 A t1 t2*
and *completely-specified M*
and *q1* ∈ *states M*
and *io* @ [(*x*, *y*)] ∈ *L A*
obtains *y''* **where** *io*@[(*x*,*y''*)] ∈ *L A* ∩ *LS M q1*
⟨*proof*⟩

lemma *separator-language-last-right* :
assumes *is-separator M q1 q2 A t1 t2*
and *completely-specified M*
and *q2* ∈ *states M*
and *io* @ [(*x*, *y*)] ∈ *L A*
obtains *y''* **where** *io*@[(*x*,*y''*)] ∈ *L A* ∩ *LS M q2*
⟨*proof*⟩

lemma *pass-separator-from-pass-io-set* :
assumes *is-separator M q1 q2 A t1 t2*
and *pass-io-set* (*from-FSM S s1*) (*atc-to-io-set* (*from-FSM M q1*) *A*)
and *observable M*
and *observable S*
and *q1* ∈ *states M*
and *s1* ∈ *states S*
and (*inputs S*) = (*inputs M*)
and *completely-specified M*
shows *pass-separator-ATC S A s1 t2*
⟨*proof*⟩

lemma *pass-separator-pass-io-set-iff*:
assumes *is-separator M q1 q2 A t1 t2*
and *observable M*
and *observable S*
and *q1* ∈ *states M*
and *s1* ∈ *states S*
and (*inputs S*) = (*inputs M*)
and *completely-specified M*
shows *pass-separator-ATC S A s1 t2* ⟷ *pass-io-set* (*from-FSM S s1*) (*atc-to-io-set* (*from-FSM M q1*) *A*)
⟨*proof*⟩

lemma *pass-separator-pass-io-set-maximal-iff*:
assumes *is-separator M q1 q2 A t1 t2*
and *observable M*
and *observable S*
and *q1* ∈ *states M*
and *s1* ∈ *states S*
and (*inputs S*) = (*inputs M*)
and *completely-specified M*
shows *pass-separator-ATC S A s1 t2* ⟷ *pass-io-set-maximal* (*from-FSM S s1*) (*remove-proper-prefixes* (*atc-to-io-set* (*from-FSM M q1*) *A*))
⟨*proof*⟩

end

36 State Preambles

This theory defines state preambles. A state preamble *P* of some state *q* of some FSM *M* is an acyclic single-input submachine of *M* that contains for each of its states and defined inputs in that state all transitions of *M* and has *q* as its only deadlock state. That is, *P* represents a strategy of reaching *q* in every complete submachine of *M*. In testing, preambles are used to reach states in the SUT that must conform to a single known state in the specification.

theory *State-Preamble*
imports ../*Product-FSM Backwards-Reachability-Analysis*
begin

definition *is-preamble* :: ('*a*,'*b*,'*c*) *fsm* ⇒ ('*a*,'*b*,'*c*) *fsm* ⇒ '*a* ⇒ *bool* **where**
is-preamble S M q =
(*acyclic S*
∧ *single-input S*

∧ *is-submachine S M*
∧ $q \in$ *reachable-states S*
∧ *deadlock-state S q*
∧ (∀ $q' \in$ *reachable-states S* .
($q = q' \vee \neg$ *deadlock-state S* q') ∧
(∀ $x \in$ *inputs M* .
(∃ $t \in$ *transitions S* . *t-source* $t = q' \wedge$ *t-input* $t = x$)
⟶ (∀ $t' \in$ *transitions M* . *t-source* $t' = q' \wedge$ *t-input* $t' = x \longrightarrow t' \in$ *transitions S*))))

fun *definitely-reachable* :: ('a,'b,'c) *fsm* ⇒ 'a ⇒ *bool* **where**
definitely-reachable M q = (∃ *S* . *is-preamble S M q*)

36.1 Basic Properties

lift-definition *initial-preamble* :: ('a,'b,'c) *fsm* ⇒ ('a,'b,'c) *fsm* **is** *FSM-Impl.initial-singleton*

⟨*proof*⟩

lemma *initial-preamble-simps*[*simp*] :
initial (*initial-preamble M*) = *initial M*
states (*initial-preamble M*) = {*initial M*}
inputs (*initial-preamble M*) = *inputs M*
outputs (*initial-preamble M*) = *outputs M*
transitions (*initial-preamble M*) = {}
⟨*proof*⟩

lemma *is-preamble-initial* :
is-preamble (*initial-preamble M*) *M* (*initial M*)
⟨*proof*⟩

lemma *is-preamble-next* :
assumes *is-preamble S M q*
and $q \neq$ *initial M*
and $t \in$ *transitions S*
and *t-source* $t =$ *initial M*
shows *is-preamble* (*from-FSM S* (*t-target t*)) (*from-FSM M* (*t-target t*)) *q*
(**is** *is-preamble ?S ?M q*)
⟨*proof*⟩

lemma *observable-preamble-paths* :
assumes *is-preamble P M* q'
and *observable M*
and *path M q p*

and *p-io p* ∈ *LS P q*
and *q* ∈ *reachable-states P*
shows *path P q p*
⟨*proof*⟩

lemma *preamble-pass-path* :
assumes *is-preamble P M q*
and $\bigwedge$ *io x y y′ . io*@[(*x*,*y*)] ∈ *L P* ⟹ *io*@[(*x*,*y′*)] ∈ *L M′* ⟹ *io*@[(*x*,*y′*)] ∈ *L P*
and *completely-specified M′*
and *inputs M′* = *inputs M*
obtains *p* **where** *path P* (*initial P*) *p* **and** *target* (*initial P*) *p* = *q* **and** *p-io p* ∈ *L M′*
⟨*proof*⟩

lemma *preamble-maximal-io-paths* :
assumes *is-preamble P M q*
and *observable M*
and *path P* (*initial P*) *p*
and *target* (*initial P*) *p* = *q*
shows ∄ *io′ . io′* ≠ [] ∧ *p-io p* @ *io′* ∈ *L P*
⟨*proof*⟩

lemma *preamble-maximal-io-paths-rev* :
assumes *is-preamble P M q*
and *observable M*
and *io* ∈ *L P*
and ∄ *io′ . io′* ≠ [] ∧ *io* @ *io′* ∈ *L P*
obtains *p* **where** *path P* (*initial P*) *p*
and *p-io p* = *io*
and *target* (*initial P*) *p* = *q*
⟨*proof*⟩

lemma *is-preamble-is-state* :
assumes *is-preamble P M q*
shows *q* ∈ *states M*
⟨*proof*⟩

36.2 Calculating State Preambles via Backwards Reachability Analysis

fun *d-states* :: (*′a*::*linorder*,*′b*::*linorder*,*′c*) *fsm* ⇒ *′a* ⇒ (*′a* × *′b*) *list* **where**
d-states M q = (*if q* = *initial M*
then []
else select-inputs (*h M*) (*initial M*) (*inputs-as-list M*) (*removeAll*

q (*removeAll* (*initial M*) (*states-as-list M*))) {*q*} [])

lemma *d-states-index-properties* :
assumes $i <$ *length* (*d-states M q*)
shows *fst* (*d-states M q* ! *i*) ∈ (*states M* − {*q*})
fst (*d-states M q* ! *i*) ≠ *q*
snd (*d-states M q* ! *i*) ∈ *inputs M*
(∀ *qx'* ∈ *set* (*take i* (*d-states M q*)) . *fst* (*d-states M q* ! *i*) ≠ *fst qx'*)
(∃ *t* ∈ *transitions M* . *t-source t* = *fst* (*d-states M q* ! *i*) ∧ *t-input t* = *snd* (*d-states M q* ! *i*))
(∀ *t* ∈ *transitions M* . (*t-source t* = *fst* (*d-states M q* ! *i*) ∧ *t-input t* = *snd* (*d-states M q* ! *i*)) ⟶ (*t-target t* = *q* ∨ (∃ *qx'* ∈ *set* (*take i* (*d-states M q*)) . *fst qx'* = (*t-target t*))))
⟨*proof*⟩

lemma *d-states-distinct* :
distinct (*map fst* (*d-states M q*))
⟨*proof*⟩

lemma *d-states-states* :
set (*map fst* (*d-states M q*)) ⊆ *states M* − {*q*}
⟨*proof*⟩

lemma *d-states-size* :
assumes *q* ∈ *states M*
shows *length* (*d-states M q*) ≤ *size M* − *1*
⟨*proof*⟩

lemma *d-states-initial* :
assumes *qx* ∈ *set* (*d-states M q*)
and *fst qx* = *initial M*
shows (*last* (*d-states M q*)) = *qx*
⟨*proof*⟩

lemma *d-states-q-noncontainment* :
shows ¬(∃ *qqx* ∈ *set* (*d-states M q*) . *fst qqx* = *q*)
⟨*proof*⟩

lemma *d-states-acyclic-paths'* :
fixes *M* :: (*'a::linorder*,*'b::linorder*,*'c*) *fsm*
assumes *path* (*filter-transitions M* (λ *t* . (*t-source t*, *t-input t*) ∈ *set* (*d-states M*

q))) *q′ p*
and *target q′ p = q′*
and *p ≠ []*
shows *False*
⟨*proof*⟩

lemma *d-states-acyclic-paths* :
fixes *M* :: (*′a*::*linorder*,*′b*::*linorder*,*′c*) *fsm*
assumes *path* (*filter-transitions M* (*λ t* . (*t-source t*, *t-input t*) ∈ *set* (*d-states M q*))) *q′ p*
(**is** *path ?FM q′ p*)
shows *distinct* (*visited-states q′ p*)
⟨*proof*⟩

lemma *d-states-induces-state-preamble-helper-acyclic* :
shows *acyclic* (*filter-transitions M* (*λ t* . (*t-source t*, *t-input t*) ∈ *set* (*d-states M q*)))
⟨*proof*⟩

lemma *d-states-induces-state-preamble-helper-single-input* :
shows *single-input* (*filter-transitions M* (*λ t* . (*t-source t*, *t-input t*) ∈ *set* (*d-states M q*)))
(**is** *single-input ?FM*)
⟨*proof*⟩

lemma *d-states-induces-state-preamble* :
assumes ∃ *qx* ∈ *set* (*d-states M q*) . *fst qx* = *initial M*
shows *is-preamble* (*filter-transitions M* (*λ t* . (*t-source t*, *t-input t*) ∈ *set* (*d-states M q*))) *M q*
(**is** *is-preamble ?S M q*)
⟨*proof*⟩

fun *calculate-state-preamble-from-input-choices* :: (*′a*::*linorder*,*′b*::*linorder*,*′c*) *fsm* ⇒ *′a* ⇒ (*′a*,*′b*,*′c*) *fsm option*
where
calculate-state-preamble-from-input-choices M q = (*if q* = *initial M*
then Some (*initial-preamble M*)
else
(*let DS* = (*d-states M q*);
DSS = *set DS*
in (*case DS of*
[] ⇒ *None* |
- ⇒ *if fst* (*last DS*) = *initial M*
then Some (*filter-transitions M* (*λ t* . (*t-source t*, *t-input t*) ∈ *DSS*))
else None)))

lemma *calculate-state-preamble-from-input-choices-soundness* :
assumes *calculate-state-preamble-from-input-choices M q = Some S*
shows *is-preamble S M q*
⟨*proof*⟩

lemma *calculate-state-preamble-from-input-choices-exhaustiveness* :
assumes ∃ *S* . *is-preamble S M q*
shows *calculate-state-preamble-from-input-choices M q* ≠ *None*
⟨*proof*⟩

36.3 Minimal Sequences to Failures extending Preambles

definition *sequence-to-failure-extending-preamble-path* ::
('a,'b,'c) *fsm* ⇒ ('d,'b,'c) *fsm* ⇒ ('a × ('a,'b,'c) *fsm*) *set* ⇒ ('a×'b×'c×'a) *list* ⇒ ('b × 'c) *list* ⇒ *bool*
where
sequence-to-failure-extending-preamble-path M M' PS p io = (∃ *q P* . *q* ∈ *states M*
∧ (*q*,*P*) ∈ *PS*
∧ *path P* (*initial P*) *p*
∧ *target* (*initial P*) *p* = *q*
∧ ((*p-io p*) @ *butlast io*) ∈ *L M*
∧ ((*p-io p*) @ *io*) ∉ *L M*
∧ ((*p-io p*) @ *io*) ∈ *L M'*)

lemma *sequence-to-failure-extending-preamble-ex* :
assumes (*initial M*, (*initial-preamble M*)) ∈ *PS* (**is** (*initial M*,*?P*) ∈ *PS*)
and ¬ *L M'* ⊆ *L M*
obtains *p io* **where** *sequence-to-failure-extending-preamble-path M M' PS p io*
⟨*proof*⟩

definition *minimal-sequence-to-failure-extending-preamble-path* ::
('a,'b,'c) *fsm* ⇒ ('d,'b,'c) *fsm* ⇒ ('a × ('a,'b,'c) *fsm*) *set* ⇒ ('a×'b×'c×'a) *list* ⇒ ('b × 'c) *list* ⇒ *bool*
where
minimal-sequence-to-failure-extending-preamble-path M M' PS p io
= ((*sequence-to-failure-extending-preamble-path M M' PS p io*)
∧ (∀ *p' io'* . *sequence-to-failure-extending-preamble-path M M' PS p' io'*
⟶ *length io* ≤ *length io'*))

lemma *minimal-sequence-to-failure-extending-preamble-ex* :
assumes (*initial M*, (*initial-preamble M*)) ∈ *PS* (**is** (*initial M*,*?P*) ∈ *PS*)
and ¬ *L M'* ⊆ *L M*

obtains *p io* **where** *minimal-sequence-to-failure-extending-preamble-path M M′ PS p io*
⟨*proof*⟩

lemma *minimal-sequence-to-failure-extending-preamble-no-repetitions-along-path* :
assumes *minimal-sequence-to-failure-extending-preamble-path M M′ PS pP io*
and *observable M*
and *path M* (*target* (*initial M*) *pP*) *p*
and *p-io p* = *butlast io*
and $q' \in$ *io-targets M′* (*p-io pP*) (*initial M′*)
and *path M′ q′ p′*
and *p-io p′* = *io*
and $i < j$
and $j <$ *length* (*butlast io*)
and $\bigwedge$ *q P*. (*q*, *P*) $\in$ *PS* $\Longrightarrow$ *is-preamble P M q*
shows *t-target* (*p* ! *i*) $\neq$ *t-target* (*p* ! *j*) $\vee$ *t-target* (*p′* ! *i*) $\neq$ *t-target* (*p′* ! *j*)
⟨*proof*⟩

lemma *minimal-sequence-to-failure-extending-preamble-no-repetitions-with-other-preambles* :
assumes *minimal-sequence-to-failure-extending-preamble-path M M′ PS pP io*
and *observable M*
and *path M* (*target* (*initial M*) *pP*) *p*
and *p-io p* = *butlast io*
and $q' \in$ *io-targets M′* (*p-io pP*) (*initial M′*)
and *path M′ q′ p′*
and *p-io p′* = *io*
and $\bigwedge$ *q P*. (*q*, *P*) $\in$ *PS* $\Longrightarrow$ *is-preamble P M q*
and $i <$ *length* (*butlast io*)
and (*t-target* (*p* ! *i*), *P′*) $\in$ *PS*
and *path P′* (*initial P′*) *pP′*
and *target* (*initial P′*) *pP′* = *t-target* (*p* ! *i*)
shows *t-target* (*p′* ! *i*) $\notin$ *io-targets M′* (*p-io pP′*) (*initial M′*)
⟨*proof*⟩

end

37 Helper Algorithms

This theory contains several algorithms used to calculate components of a test suite.

theory *Helper-Algorithms*
imports *State-Separator State-Preamble*
begin

37.1 Calculating r-distinguishable State Pairs with Separators

```
definition r-distinguishable-state-pairs-with-separators ::
  ('a::linorder,'b::linorder,'c) fsm ⇒ (('a × 'a) × (('a × 'a) + 'a,'b,'c) fsm) set
  where
  r-distinguishable-state-pairs-with-separators M =
    {((q1,q2),Sep) | q1 q2 Sep . q1 ∈ states M
                    ∧ q2 ∈ states M
                     ∧ ((q1 < q2 ∧ state-separator-from-s-states M q1 q2 = Some
Sep)
                      ∨ (q2 < q1 ∧ state-separator-from-s-states M q2 q1 = Some
Sep)) }

lemma r-distinguishable-state-pairs-with-separators-alt-def :
  r-distinguishable-state-pairs-with-separators M =
    ⋃ (image (λ ((q1,q2),A) . {((q1,q2),the A),((q2,q1),the A)})
          (Set.filter (λ (qq,A) . A ≠ None)
                (image (λ (q1,q2) . ((q1,q2),state-separator-from-s-states M
q1 q2))
                      (Set.filter (λ (q1,q2) . q1 < q2) (states M × states
M)))))
  (is ?P1 = ?P2)
⟨proof⟩


lemma r-distinguishable-state-pairs-with-separators-code[code] :
  r-distinguishable-state-pairs-with-separators M =
    set (concat (map
          (λ ((q1,q2),A) . [((q1,q2),the A),((q2,q1),the A)])
          (filter (λ (qq,A) . A ≠ None)
                (map (λ (q1,q2) . ((q1,q2),state-separator-from-s-states M q1
q2))
                    (filter (λ (q1,q2) . q1 < q2)
                        (List.product(states-as-list M) (states-as-list M)))))))
  (is r-distinguishable-state-pairs-with-separators M = ?C2)
⟨proof⟩


lemma r-distinguishable-state-pairs-with-separators-same-pair-same-separator :
  assumes ((q1,q2),A) ∈ r-distinguishable-state-pairs-with-separators M
  and     ((q1,q2),A') ∈ r-distinguishable-state-pairs-with-separators M
shows A = A'
  ⟨proof⟩


lemma r-distinguishable-state-pairs-with-separators-sym-pair-same-separator :
  assumes ((q1,q2),A) ∈ r-distinguishable-state-pairs-with-separators M
  and     ((q2,q1),A') ∈ r-distinguishable-state-pairs-with-separators M
shows A = A'
```

⟨*proof*⟩

lemma *r-distinguishable-state-pairs-with-separators-elem-is-separator*:
assumes ((*q1*,*q2*),*A*) ∈ *r-distinguishable-state-pairs-with-separators M*
and *observable M*
and *completely-specified M*
shows *is-separator M q1 q2 A* (*Inr q1*) (*Inr q2*)
⟨*proof*⟩

37.2 Calculating Pairwise r-distinguishable Sets of States

definition *pairwise-r-distinguishable-state-sets-from-separators* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*) *fsm* ⇒ '*a set set* **where**
pairwise-r-distinguishable-state-sets-from-separators M
= { *S* . *S* ⊆ *states M* ∧ (∀ *q1* ∈ *S* . ∀ *q2* ∈ *S* . *q1* ≠ *q2* ⟶ (*q1*,*q2*) ∈ *image fst* (*r-distinguishable-state-pairs-with-separators M*))}

definition *pairwise-r-distinguishable-state-sets-from-separators-list* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*) *fsm* ⇒ '*a set list* **where**
pairwise-r-distinguishable-state-sets-from-separators-list M =
(*let RDS* = *image fst* (*r-distinguishable-state-pairs-with-separators M*)
in filter (λ *S* . ∀ *q1* ∈ *S* . ∀ *q2* ∈ *S* . *q1* ≠ *q2* ⟶ (*q1*,*q2*) ∈ *RDS*)
(*map set* (*pow-list* (*states-as-list M*))))

lemma *pairwise-r-distinguishable-state-sets-from-separators-code*[*code*] :
pairwise-r-distinguishable-state-sets-from-separators M = *set* (*pairwise-r-distinguishable-state-sets-from-separ*
M)
⟨*proof*⟩

lemma *pairwise-r-distinguishable-state-sets-from-separators-cover* :
assumes *q* ∈ *states M*
shows ∃ *S* ∈ (*pairwise-r-distinguishable-state-sets-from-separators M*) . *q* ∈ *S*
⟨*proof*⟩

definition *maximal-pairwise-r-distinguishable-state-sets-from-separators* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*) *fsm* ⇒ '*a set set* **where**
maximal-pairwise-r-distinguishable-state-sets-from-separators M
= { *S* . *S* ∈ (*pairwise-r-distinguishable-state-sets-from-separators M*)
∧ (∄ *S'* . *S'* ∈ (*pairwise-r-distinguishable-state-sets-from-separators M*) ∧ *S* ⊂ *S'*)}

definition *maximal-pairwise-r-distinguishable-state-sets-from-separators-list* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*) *fsm* ⇒ '*a set list* **where**

maximal-pairwise-r-distinguishable-state-sets-from-separators-list M =
remove-subsets (*pairwise-r-distinguishable-state-sets-from-separators-list M*)

lemma *maximal-pairwise-r-distinguishable-state-sets-from-separators-code*[*code*] :
maximal-pairwise-r-distinguishable-state-sets-from-separators M
= *set* (*maximal-pairwise-r-distinguishable-state-sets-from-separators-list M*)
⟨*proof*⟩

lemma *maximal-pairwise-r-distinguishable-state-sets-from-separators-cover* :
assumes $q \in$ *states M*
shows $\exists$ $S \in$ (*maximal-pairwise-r-distinguishable-state-sets-from-separators M*).
$q \in S$
⟨*proof*⟩

37.3 Calculating d-reachable States with Preambles

definition *d-reachable-states-with-preambles* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*) *fsm* ⇒
('*a* × ('*a*,'*b*,'*c*) *fsm*) *set* **where**
d-reachable-states-with-preambles M =
image (λ *qp* . (*fst qp*, *the* (*snd qp*)))
(*Set.filter* (λ *qp* . *snd qp* ≠ *None*)
(*image* (λ *q* . (*q*, *calculate-state-preamble-from-input-choices M*
q))
(*states M*)))

lemma *d-reachable-states-with-preambles-exhaustiveness* :
assumes ∃ *P* . *is-preamble P M q*
and $q \in$ *states M*
shows ∃ *P* . (*q*,*P*) ∈ (*d-reachable-states-with-preambles M*)
⟨*proof*⟩

lemma *d-reachable-states-with-preambles-soundness* :
assumes (*q*,*P*) ∈ (*d-reachable-states-with-preambles M*)
and *observable M*
shows *is-preamble P M q*
and $q \in$ *states M*
⟨*proof*⟩

37.4 Calculating Repetition Sets

Repetition sets are sets of tuples each containing a maximal set of pairwise r-distinguishable states and the subset of those states that have a preamble.

definition *maximal-repetition-sets-from-separators* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*)
fsm ⇒ ('*a set* × '*a set*) *set* **where**

maximal-repetition-sets-from-separators M
  $= \{(S, S \cap (image\ fst\ (d\text{-}reachable\text{-}states\text{-}with\text{-}preambles\ M))) \mid S\ .$
    $S \in (maximal\text{-}pairwise\text{-}r\text{-}distinguishable\text{-}state\text{-}sets\text{-}from\text{-}separators\ M)\}$

definition *maximal-repetition-sets-from-separators-list-naive* :: ('a::*linorder*,'b::*linorder*,'c) *fsm* ⇒ ('a *set* × 'a *set*) *list* **where**
  maximal-repetition-sets-from-separators-list-naive M
  = (*let DR* = (*image fst* (*d-reachable-states-with-preambles M*))
    in map (λ *S* . (*S*, *S* ∩ *DR*)) (*maximal-pairwise-r-distinguishable-state-sets-from-separators-list M*))

lemma *maximal-repetition-sets-from-separators-code*[*code*]:
  maximal-repetition-sets-from-separators M = (*let DR* = (*image fst* (*d-reachable-states-with-preambles M*))
    in image (λ *S* . (*S*, *S* ∩ *DR*)) (*maximal-pairwise-r-distinguishable-state-sets-from-separators M*))
  ⟨*proof*⟩

lemma *maximal-repetition-sets-from-separators-code-alt*:
  maximal-repetition-sets-from-separators M = *set* (*maximal-repetition-sets-from-separators-list-naive M*)
  ⟨*proof*⟩

37.4.1 Calculating Sub-Optimal Repetition Sets

Finding maximal pairwise r-distinguishable subsets of the state set of some FSM is likely too expensive for FSMs containing a large number of r-distinguishable pairs of states. The following functions calculate only subset of all repetition sets while maintaining the property that every state is contained in some repetition set.

fun *extend-until-conflict* :: ('a × 'a) *set* ⇒ 'a *list* ⇒ 'a *list* ⇒ *nat* ⇒ 'a *list* **where**
  extend-until-conflict non-confl-set candidates xs 0 = *xs* |
  extend-until-conflict non-confl-set candidates xs (*Suc k*) = (*case dropWhile* (λ *x* . *find* (λ *y* . (*x*,*y*) ∉ *non-confl-set*) *xs* ≠ *None*) *candidates of*
    [] ⇒ *xs* |
    (*c*#*cs*) ⇒ *extend-until-conflict non-confl-set cs* (*c*#*xs*) *k*)

lemma *extend-until-conflict-retainment* :
  assumes *x* ∈ *set xs*
  shows *x* ∈ *set* (*extend-until-conflict non-confl-set candidates xs k*)
⟨*proof*⟩

lemma *extend-until-conflict-elem* :
  assumes *x* ∈ *set* (*extend-until-conflict non-confl-set candidates xs k*)
  shows *x* ∈ *set xs* ∨ *x* ∈ *set candidates*
⟨*proof*⟩

lemma *extend-until-conflict-no-conflicts* :
assumes $x \in$ *set* (*extend-until-conflict non-confl-set candidates xs k*)
and $y \in$ *set* (*extend-until-conflict non-confl-set candidates xs k*)
and $x \in$ *set xs* $\Longrightarrow y \in$ *set xs* $\Longrightarrow (x,y) \in$ *non-confl-set* $\vee$ $(y,x) \in$ *non-confl-set*

and $x \neq y$
shows $(x,y) \in$ *non-confl-set* $\vee$ $(y,x) \in$ *non-confl-set*
⟨*proof*⟩

definition *greedy-pairwise-r-distinguishable-state-sets-from-separators* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*) *fsm* ⇒ '*a set list* **where**
greedy-pairwise-r-distinguishable-state-sets-from-separators M =
(*let pwrds* = *image fst* (*r-distinguishable-state-pairs-with-separators M*);
k = *size M*;
nL = *states-as-list M*
in map (λ*q* . *set* (*extend-until-conflict pwrds* (*remove1 q nL*) [*q*] *k*)) *nL*)

definition *maximal-repetition-sets-from-separators-list-greedy* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*) *fsm* ⇒ ('*a set* × '*a set*) *list* **where**
maximal-repetition-sets-from-separators-list-greedy M = (*let DR* = (*image fst* (*d-reachable-states-with-preambles M*))
in remdups (*map* (λ *S* . (*S*, *S* ∩ *DR*)) (*greedy-pairwise-r-distinguishable-state-sets-from-separators M*)))

lemma *greedy-pairwise-r-distinguishable-state-sets-from-separators-cover* :
assumes $q \in$ *states M*
shows ∃ *S* ∈ *set* (*greedy-pairwise-r-distinguishable-state-sets-from-separators M*). *q* ∈ *S*
⟨*proof*⟩

lemma *r-distinguishable-state-pairs-with-separators-sym* :
assumes (*q1*,*q2*) ∈ *fst* ' *r-distinguishable-state-pairs-with-separators M*
shows (*q2*,*q1*) ∈ *fst* ' *r-distinguishable-state-pairs-with-separators M*
⟨*proof*⟩

lemma *greedy-pairwise-r-distinguishable-state-sets-from-separators-soundness* :
set (*greedy-pairwise-r-distinguishable-state-sets-from-separators M*) ⊆ (*pairwise-r-distinguishable-state-sets-fr M*)
⟨*proof*⟩

end

38 Maximal Path Tries

Drastically reduced implementation of tries that consider only maximum length sequences as elements. Inserting a sequence that is prefix of some already contained sequence does not alter the trie. Intended to store IO-sequences to apply in testing, as in this use-case proper prefixes need not be applied separately.

theory *Maximal-Path-Trie*
imports ../*Util*
begin

38.1 Utils for Updating Associative Lists

fun *update-assoc-list-with-default* :: *'a* ⇒ (*'b* ⇒ *'b*) ⇒ *'b* ⇒ (*'a* × *'b*) *list* ⇒ (*'a* × *'b*) *list* **where**
update-assoc-list-with-default k f d [] = [(*k*,*f d*)] |
update-assoc-list-with-default k f d ((*x*,*y*)#*xys*) = (*if k* = *x*
then ((*x*,*f y*)#*xys*)
else (*x*,*y*) # (*update-assoc-list-with-default k f d xys*))

lemma *update-assoc-list-with-default-key-found* :
assumes *distinct* (*map fst xys*)
and *i* < *length xys*
and *fst* (*xys* ! *i*) = *k*
shows *update-assoc-list-with-default k f d xys* =
((*take i xys*) @ [(*k*, *f* (*snd* (*xys* ! *i*)))] @ (*drop* (*Suc i*) *xys*))
⟨*proof*⟩

lemma *update-assoc-list-with-default-key-not-found* :
assumes *distinct* (*map fst xys*)
and *k* ∉ *set* (*map fst xys*)
shows *update-assoc-list-with-default k f d xys* = *xys* @ [(*k*,*f d*)]
⟨*proof*⟩

lemma *update-assoc-list-with-default-key-distinct* :
assumes *distinct* (*map fst xys*)
shows *distinct* (*map fst* (*update-assoc-list-with-default k f d xys*))
⟨*proof*⟩

38.2 Maximum Path Trie Implementation

datatype *'a mp-trie* = *MP-Trie* (*'a* × *'a mp-trie*) *list*

fun *mp-trie-invar* :: *'a mp-trie* ⇒ *bool* **where**
mp-trie-invar (*MP-Trie ts*) = (*distinct* (*map fst ts*) ∧ (∀ *t* ∈ *set* (*map snd ts*) . *mp-trie-invar t*))

definition *empty* :: *'a mp-trie* **where**
empty = *MP-Trie* []

lemma *empty-invar* : *mp-trie-invar empty* ⟨*proof*⟩

fun *height* :: *'a mp-trie* ⇒ *nat* **where**
height (*MP-Trie* []) = *0* |
height (*MP-Trie* (*xt#xts*)) = *Suc* (*foldr* (λ *t m* . *max* (*height t*) *m*) (*map snd* (*xt#xts*)) *0*)

lemma *height-0* :
assumes *height t* = *0*
shows *t* = *empty*
⟨*proof*⟩

lemma *height-inc* :
assumes *t* ∈ *set* (*map snd ts*)
shows *height t* < *height* (*MP-Trie ts*)
⟨*proof*⟩

fun *insert* :: *'a list* ⇒ *'a mp-trie* ⇒ *'a mp-trie* **where**
insert [] *t* = *t* |
insert (*x#xs*) (*MP-Trie ts*) = (*MP-Trie* (*update-assoc-list-with-default x* (λ *t* . *insert xs t*) *empty ts*))

lemma *insert-invar* : *mp-trie-invar t* ⟹ *mp-trie-invar* (*insert xs t*)
⟨*proof*⟩

fun *paths* :: *'a mp-trie* ⇒ *'a list list* **where**
paths (*MP-Trie* []) = [[]] |
paths (*MP-Trie* (*t#ts*)) = *concat* (*map* (λ (*x*,*t*) . *map* ((#) *x*) (*paths t*)) (*t#ts*))

lemma *paths-empty* :
assumes [] ∈ *set* (*paths t*)
shows *t* = *empty*
⟨*proof*⟩

lemma *paths-nonempty* :

assumes $[] \notin$ *set* (*paths t*)
shows *set* (*paths t*) $\neq$ {}
⟨*proof*⟩

lemma *paths-maximal*: *mp-trie-invar t* $\Longrightarrow$ *xs'* $\in$ *set* (*paths t*) $\Longrightarrow$ $\neg$ ($\exists$ *xs''* . *xs''* $\neq$ [] $\wedge$ *xs'@xs''* $\in$ *set* (*paths t*))
⟨*proof*⟩

lemma *paths-insert-empty* :
paths (*insert xs empty*) = [*xs*]
⟨*proof*⟩

lemma *paths-order* :
assumes *set ts* = *set ts'*
and *length ts* = *length ts'*
shows *set* (*paths* (*MP-Trie ts*)) = *set* (*paths* (*MP-Trie ts'*))
⟨*proof*⟩

lemma *paths-insert-maximal* :
assumes *mp-trie-invar t*
shows *set* (*paths* (*insert xs t*)) = (*if* ($\exists$ *xs'* . *xs@xs'* $\in$ *set* (*paths t*))
then set (*paths t*)
else Set.insert xs (*set* (*paths t*) $-$ {*xs'* . $\exists$ *xs''* . *xs'@xs''* = *xs*}))
⟨*proof*⟩

fun *from-list* :: *'a list list* $\Rightarrow$ *'a mp-trie* **where**
from-list seqs = *foldr insert seqs empty*

lemma *from-list-invar* : *mp-trie-invar* (*from-list xs*)
⟨*proof*⟩

lemma *from-list-paths* :
set (*paths* (*from-list* (*x#xs*))) = {*y*. *y* $\in$ *set* (*x#xs*) $\wedge$ $\neg$($\exists$ *y'* . *y'* $\neq$ [] $\wedge$ *y@y'* $\in$ *set* (*x#xs*))}
⟨*proof*⟩

38.2.1 New Code Generation for *remove-proper-prefixes*

declare [[*code drop*: *remove-proper-prefixes*]]

lemma *remove-proper-prefixes-code-trie*[*code*] :

remove-proper-prefixes (*set xs*) = (*case xs of* [] ⇒ {} | (*x#xs′*) ⇒ *set* (*paths* (*from-list* (*x#xs′*))))
⟨*proof*⟩

end

39 R-Distinguishability

This theory defines the notion of r-distinguishability and relates it to state separators.

theory *R-Distinguishability*
imports *State-Separator*
begin

definition *r-compatible* :: (′*a*, ′*b*, ′*c*) *fsm* ⇒ ′*a* ⇒ ′*a* ⇒ *bool* **where**
r-compatible M q1 q2 = ((∃ *S* . *completely-specified S* ∧ *is-submachine S* (*product* (*from-FSM M q1*) (*from-FSM M q2*))))

abbreviation(*input*) *r-distinguishable M q1 q2* ≡ ¬ *r-compatible M q1 q2*

fun *r-distinguishable-k* :: (′*a*, ′*b*, ′*c*) *fsm* ⇒ ′*a* ⇒ ′*a* ⇒ *nat* ⇒ *bool* **where**
r-distinguishable-k M q1 q2 0 = (∃ *x* ∈ (*inputs M*) . ¬ (∃ *t1* ∈ *transitions M* . ∃ *t2* ∈ *transitions M* . *t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2*)) |
r-distinguishable-k M q1 q2 (*Suc k*) = (*r-distinguishable-k M q1 q2 k*
∨ (∃ *x* ∈ (*inputs M*) . ∀ *t1* ∈ *transitions M* . ∀ *t2* ∈ *transitions M* . (*t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2*) ⟶ *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k*))

39.1 R(k)-Distinguishability Properties

lemma *r-distinguishable-k-0-alt-def* :
r-distinguishable-k M q1 q2 0 = (∃ *x* ∈ (*inputs M*) . ¬(∃ *y q1′ q2′* . (*q1*,*x*,*y*,*q1′*) ∈ *transitions M* ∧ (*q2*,*x*,*y*,*q2′*) ∈ *transitions M*))
⟨*proof*⟩

lemma *r-distinguishable-k-Suc-k-alt-def* :
r-distinguishable-k M q1 q2 (*Suc k*) = (*r-distinguishable-k M q1 q2 k*
∨ (∃ *x* ∈ (*inputs M*) . ∀ *y q1′ q2′* . ((*q1*,*x*,*y*,*q1′*) ∈ *transitions M* ∧ (*q2*,*x*,*y*,*q2′*) ∈ *transitions M*) ⟶ *r-distinguishable-k M q1′ q2′ k*))
⟨*proof*⟩

lemma *r-distinguishable-k-by-larger* :
assumes *r-distinguishable-k M q1 q2 k*
and $k \leq k'$
shows *r-distinguishable-k M q1 q2 k′*
⟨*proof*⟩

lemma *r-distinguishable-k-0-not-completely-specified* :
assumes *r-distinguishable-k M q1 q2 0*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows ¬ *completely-specified-state* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*initial* (*product* (*from-FSM M q1*) (*from-FSM M q2*)))
⟨*proof*⟩

lemma *r-0-distinguishable-from-not-completely-specified-initial* :
assumes ¬ *completely-specified-state* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*q1*,*q2*)
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *r-distinguishable-k M q1 q2 0*
⟨*proof*⟩

lemma *r-0-distinguishable-from-not-completely-specified* :
assumes ¬ *completely-specified-state* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*q1′*,*q2′*)
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and (*q1′*,*q2′*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
shows *r-distinguishable-k M q1′ q2′ 0*
⟨*proof*⟩

lemma *r-distinguishable-k-intersection-path* :
assumes ¬ *r-distinguishable-k M q1 q2 k*
and *length xs* ≤ *Suc k*
and *set xs* ⊆ (*inputs M*)
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows ∃ *p* . *path* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*q1*,*q2*) *p* ∧ *map fst* (*p-io p*) = *xs*
⟨*proof*⟩

lemma *r-distinguishable-k-intersection-paths* :
assumes ¬(∃ *k* . *r-distinguishable-k M q1 q2 k*)

and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows ∀ *xs* . *set xs* ⊆ (*inputs M*) ⟶ (∃ *p* . *path* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*q1*,*q2*) *p* ∧ *map fst* (*p-io p*) = *xs*)
⟨*proof*⟩

39.1.1 Equivalence of R-Distinguishability Definitions

lemma *r-distinguishable-alt-def* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows *r-distinguishable M q1 q2* ⟷ (∃ *k* . *r-distinguishable-k M q1 q2 k*)
⟨*proof*⟩

39.2 Bounds

inductive *is-least-r-d-k-path* :: (′*a*, ′*b*, ′*c*) *fsm* ⇒ ′*a* ⇒ ′*a* ⇒ ((′*a* × ′*a*) × ′*b* × *nat*) *list* ⇒ *bool* **where**
immediate[*intro*!] : *x* ∈ (*inputs M*) ⟹ ¬ (∃ *t1* ∈ *transitions M* . ∃ *t2* ∈ *transitions M* . *t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2*) ⟹ *is-least-r-d-k-path M q1 q2* [((*q1*,*q2*),*x*,*0*)] |
step[*intro*!] : *Suc k* = (*LEAST k*′ . *r-distinguishable-k M q1 q2 k*′)
⟹ *x* ∈ (*inputs M*)
⟹ (∀ *t1* ∈ *transitions M* . ∀ *t2* ∈ *transitions M* . (*t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2*) ⟶ *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k*)
⟹ *t1* ∈ *transitions M*
⟹ *t2* ∈ *transitions M*
⟹ *t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2*
⟹ *is-least-r-d-k-path M* (*t-target t1*) (*t-target t2*) *p*
⟹ *is-least-r-d-k-path M q1 q2* (((*q1*,*q2*),*x*,*Suc k*)#*p*)

inductive-cases *is-least-r-d-k-path-immediate-elim*[*elim*!]: *is-least-r-d-k-path M q1 q2* [((*q1*,*q2*),*x*,*0*)]
inductive-cases *is-least-r-d-k-path-step-elim*[*elim*!]: *is-least-r-d-k-path M q1 q2* (((*q1*,*q2*),*x*,*Suc k*)#*p*)

lemma *is-least-r-d-k-path-nonempty* :
assumes *is-least-r-d-k-path M q1 q2 p*
shows *p* ≠ []
⟨*proof*⟩

lemma *is-least-r-d-k-path-0-extract* :
assumes *is-least-r-d-k-path M q1 q2* [*t*]
shows ∃ *x* . *t* = ((*q1*,*q2*),*x*,*0*)
⟨*proof*⟩

lemma *is-least-r-d-k-path-Suc-extract* :
assumes *is-least-r-d-k-path M q1 q2* (*t*#*t*′#*p*)

shows $\exists\ x\ k\ .\ t = ((q1,q2),x,Suc\ k)$
⟨*proof*⟩

lemma *is-least-r-d-k-path-Suc-transitions* :
assumes *is-least-r-d-k-path M q1 q2* $(((q1,q2),x,Suc\ k)\#p)$
shows $(\forall\ t1 \in transitions\ M\ .\ \forall\ t2 \in transitions\ M\ .\ (t\text{-}source\ t1 = q1 \wedge t\text{-}source\ t2 = q2 \wedge t\text{-}input\ t1 = x \wedge t\text{-}input\ t2 = x \wedge t\text{-}output\ t1 = t\text{-}output\ t2) \longrightarrow r\text{-}distinguishable\text{-}k\ M\ (t\text{-}target\ t1)\ (t\text{-}target\ t2)\ k)$
⟨*proof*⟩

lemma *is-least-r-d-k-path-is-least* :
assumes *is-least-r-d-k-path M q1 q2* $(t\#p)$
shows $r\text{-}distinguishable\text{-}k\ M\ q1\ q2\ (snd\ (snd\ t)) \wedge (snd\ (snd\ t)) = (LEAST\ k'\ .\ r\text{-}distinguishable\text{-}k\ M\ q1\ q2\ k')$
⟨*proof*⟩

lemma *r-distinguishable-k-least-next* :
assumes $\exists\ k\ .\ r\text{-}distinguishable\text{-}k\ M\ q1\ q2\ k$
and $(LEAST\ k\ .\ r\text{-}distinguishable\text{-}k\ M\ q1\ q2\ k) = Suc\ k$
and $x \in (inputs\ M)$
and $\forall\, t1 \in transitions\ M.\ \forall\, t2 \in transitions\ M.$
$t\text{-}source\ t1 = q1 \wedge$
$t\text{-}source\ t2 = q2 \wedge t\text{-}input\ t1 = x \wedge t\text{-}input\ t2 = x \wedge t\text{-}output\ t1 = t\text{-}output\ t2 \longrightarrow$
$r\text{-}distinguishable\text{-}k\ M\ (t\text{-}target\ t1)\ (t\text{-}target\ t2)\ k$
shows $\exists\ t1 \in transitions\ M\ .\ \exists\ t2 \in transitions\ M\ .\ (t\text{-}source\ t1 = q1 \wedge t\text{-}source\ t2 = q2 \wedge t\text{-}input\ t1 = x \wedge t\text{-}input\ t2 = x \wedge t\text{-}output\ t1 = t\text{-}output\ t2) \wedge (LEAST\ k\ .\ r\text{-}distinguishable\text{-}k\ M\ (t\text{-}target\ t1)\ (t\text{-}target\ t2)\ k) = k$
⟨*proof*⟩

lemma *is-least-r-d-k-path-length-from-r-d* :
assumes $\exists\ k\ .\ r\text{-}distinguishable\text{-}k\ M\ q1\ q2\ k$
shows $\exists\ t\ p\ .\ is\text{-}least\text{-}r\text{-}d\text{-}k\text{-}path\ M\ q1\ q2\ (t\#p) \wedge length\ (t\#p) = Suc\ (LEAST\ k\ .\ r\text{-}distinguishable\text{-}k\ M\ q1\ q2\ k)$
⟨*proof*⟩

lemma *is-least-r-d-k-path-states* :
assumes *is-least-r-d-k-path M q1 q2 p*
and $q1 \in states\ M$
and $q2 \in states\ M$
shows $set\ (map\ fst\ p) \subseteq states\ (product\ (from\text{-}FSM\ M\ q1)\ (from\text{-}FSM\ M\ q2))$
⟨*proof*⟩

lemma *is-least-r-d-k-path-decreasing* :
assumes *is-least-r-d-k-path M q1 q2 p*
shows ∀ *t′* ∈ *set* (*tl p*) . *snd* (*snd t′*) < *snd* (*snd* (*hd p*))
⟨*proof*⟩

lemma *is-least-r-d-k-path-suffix* :
assumes *is-least-r-d-k-path M q1 q2 p*
and *i* < *length p*
shows *is-least-r-d-k-path M* (*fst* (*fst* (*hd* (*drop i p*)))) (*snd* (*fst* (*hd* (*drop i p*)))) (*drop i p*)
⟨*proof*⟩

lemma *is-least-r-d-k-path-distinct* :
assumes *is-least-r-d-k-path M q1 q2 p*
shows *distinct* (*map fst p*)
⟨*proof*⟩

lemma *r-distinguishable-k-least-bound* :
assumes ∃ *k* . *r-distinguishable-k M q1 q2 k*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows (*LEAST k* . *r-distinguishable-k M q1 q2 k*) ≤ (*size* (*product* (*from-FSM M q1*) (*from-FSM M q2*)))
⟨*proof*⟩

39.3 Deciding R-Distinguishability

fun *r-distinguishable-k-least* :: (*′a*, *′b*::*linorder*, *′c*) *fsm* ⇒ *′a* ⇒ *′a* ⇒ *nat* ⇒ (*nat* × *′b*) *option* **where**
r-distinguishable-k-least M q1 q2 0 = (*case find* (λ *x* . ¬ (∃ *t1* ∈ *transitions M* . ∃ *t2* ∈ *transitions M* . *t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2*)) (*sort* (*inputs-as-list M*)) *of*
Some x ⇒ *Some* (*0*,*x*) |
None ⇒ *None*) |
r-distinguishable-k-least M q1 q2 (*Suc n*) = (*case r-distinguishable-k-least M q1 q2 n of*
Some k ⇒ *Some k* |
None ⇒ (*case find* (λ *x* . ∀ *t1* ∈ *transitions M* . ∀ *t2* ∈ *transitions M* . (*t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2*) ⟶ *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *n*) (*sort* (*inputs-as-list M*)) *of*
Some x ⇒ *Some* (*Suc n*,*x*) |
None ⇒ *None*))

lemma *r-distinguishable-k-least-ex* :

assumes *r-distinguishable-k-least M q1 q2 k = None*
shows ¬ *r-distinguishable-k M q1 q2 k*
⟨*proof*⟩

lemma *r-distinguishable-k-least-0-correctness* :
assumes *r-distinguishable-k-least M q1 q2 n = Some (0,x)*
shows *r-distinguishable-k M q1 q2 0* ∧ *0 =*
(*LEAST k . r-distinguishable-k M q1 q2 k*)
∧ (*x* ∈ (*inputs M*) ∧ ¬ (∃ *t1* ∈ *transitions M* . ∃ *t2* ∈ *transitions M* . *t-source t1 = q1* ∧ *t-source t2 = q2* ∧ *t-input t1 = x* ∧ *t-input t2 = x* ∧ *t-output t1 = t-output t2*))
∧ (∀ *x′* ∈ (*inputs M*) . *x′ < x* ⟶ (∃ *t1* ∈ *transitions M* . ∃ *t2* ∈ *transitions M* . *t-source t1 = q1* ∧ *t-source t2 = q2* ∧ *t-input t1 = x′* ∧ *t-input t2 = x′* ∧ *t-output t1 = t-output t2*))
⟨*proof*⟩

lemma *r-distinguishable-k-least-Suc-correctness* :
assumes *r-distinguishable-k-least M q1 q2 n = Some (Suc k,x)*
shows *r-distinguishable-k M q1 q2* (*Suc k*) ∧ (*Suc k*) =
(*LEAST k . r-distinguishable-k M q1 q2 k*)
∧ (*x* ∈ (*inputs M*) ∧ (∀ *t1* ∈ *transitions M* . ∀ *t2* ∈ *transitions M* . (*t-source t1 = q1* ∧ *t-source t2 = q2* ∧ *t-input t1 = x* ∧ *t-input t2 = x* ∧ *t-output t1 = t-output t2*) ⟶ *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k*))
∧ (∀ *x′* ∈ (*inputs M*) . *x′ < x* ⟶ ¬(∀ *t1* ∈ *transitions M* . ∀ *t2* ∈ *transitions M* . (*t-source t1 = q1* ∧ *t-source t2 = q2* ∧ *t-input t1 = x′* ∧ *t-input t2 = x′* ∧ *t-output t1 = t-output t2*) ⟶ *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k*))
⟨*proof*⟩

lemma *r-distinguishable-k-least-is-least* :
assumes *r-distinguishable-k-least M q1 q2 n = Some (k,x)*
shows (∃ *k* . *r-distinguishable-k M q1 q2 k*) ∧ (*k* = (*LEAST k . r-distinguishable-k M q1 q2 k*))
⟨*proof*⟩

lemma *r-distinguishable-k-from-r-distinguishable-k-least* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows (∃ *k* . *r-distinguishable-k M q1 q2 k*) = (*r-distinguishable-k-least M q1 q2* (*size* (*product* (*from-FSM M q1*) (*from-FSM M q2*))) ≠ *None*)
(**is** *?P1 = ?P2*)
⟨*proof*⟩

definition *is-r-distinguishable* :: (*′a, ′b, ′c*) *fsm* ⇒ *′a* ⇒ *′a* ⇒ *bool* **where**
is-r-distinguishable M q1 q2 = (∃ *k* . *r-distinguishable-k M q1 q2 k*)

lemma *is-r-distinguishable-contained-code*[*code*] :
is-r-distinguishable M q1 q2 = (*if* (*q1* ∈ *states M* ∧ *q2* ∈ *states M*) *then* (*r-distinguishable-k-least M q1 q2* (*size* (*product* (*from-FSM M q1*) (*from-FSM M q2*))) ≠ *None*)
else ¬(*inputs M* = {}))
⟨*proof*⟩

39.4 State Separators and R-Distinguishability

lemma *state-separator-r-distinguishes-k* :
assumes *is-state-separator-from-canonical-separator* (*canonical-separator M q1 q2*) *q1 q2 S*
and *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows ∃ *k* . *r-distinguishable-k M q1 q2 k*
⟨*proof*⟩

end

40 Traversal Set

This theory defines the calculation of m-traversal paths. These are paths extended from some state until they visit pairwise r-distinguishable states a number of times dependent on m.

theory *Traversal-Set*
imports *Helper-Algorithms*
begin

definition *m-traversal-paths-with-witness-up-to-length* ::
('*a*,'*b*,'*c*) *fsm* ⇒ '*a* ⇒ ('*a set* × '*a set*) *list* ⇒ *nat* ⇒ *nat* ⇒ (('*a*×'*b*×'*c*×'*a*) *list* × ('*a set* × '*a set*)) *set*
where
m-traversal-paths-with-witness-up-to-length M q D m k
= *paths-up-to-length-or-condition-with-witness M* (λ *p* . *find* (λ *d* . *length* (*filter* (λ*t* . *t-target t* ∈ *fst d*) *p*) ≥ *Suc* (*m* − (*card* (*snd d*)))) *D*) *k q*

definition *m-traversal-paths-with-witness* ::
('*a*,'*b*,'*c*) *fsm* ⇒ '*a* ⇒ ('*a set* × '*a set*) *list* ⇒ *nat* ⇒ (('*a*×'*b*×'*c*×'*a*) *list* × ('*a set* × '*a set*)) *set*
where
m-traversal-paths-with-witness M q D m = *m-traversal-paths-with-witness-up-to-length M q D m* (*Suc* (*size M* ∗ *m*))

lemma *m-traversal-paths-with-witness-finite* : *finite* (*m-traversal-paths-with-witness M q D m*)

⟨*proof*⟩

lemma *m-traversal-paths-with-witness-up-to-length-max-length* :
assumes $\bigwedge q . q \in states\ M \Longrightarrow \exists\ d \in set\ D . q \in fst\ d$
and $\bigwedge d . d \in set\ D \Longrightarrow snd\ d \subseteq fst\ d$
and $q \in states\ M$
and $(p,d) \in (m\text{-}traversal\text{-}paths\text{-}with\text{-}witness\text{-}up\text{-}to\text{-}length\ M\ q\ D\ m\ k)$
shows $length\ p \leq Suc\ ((size\ M) * m)$
⟨*proof*⟩

lemma *m-traversal-paths-with-witness-set* :
assumes $\bigwedge q . q \in states\ M \Longrightarrow \exists\ d \in set\ D . q \in fst\ d$
and $\bigwedge d . d \in set\ D \Longrightarrow snd\ d \subseteq fst\ d$
and $q \in states\ M$
shows $(m\text{-}traversal\text{-}paths\text{-}with\text{-}witness\ M\ q\ D\ m)$
$= \{(p,d) \mid p\ d . path\ M\ q\ p$
$\wedge find\ (\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p))\ D = Some\ d$
$\wedge (\forall p'\ p''.\ p = p' @ p'' \wedge p'' \neq [] \longrightarrow find\ (\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p'))\ D = None)\}$
(**is** *?MTP* = *?P*)
⟨*proof*⟩

lemma *maximal-repetition-sets-from-separators-cover* :
assumes $q \in states\ M$
shows $\exists\ d \in (maximal\text{-}repetition\text{-}sets\text{-}from\text{-}separators\ M) . q \in fst\ d$
⟨*proof*⟩

lemma *maximal-repetition-sets-from-separators-d-reachable-subset* :
shows $\bigwedge d . d \in (maximal\text{-}repetition\text{-}sets\text{-}from\text{-}separators\ M) \Longrightarrow snd\ d \subseteq fst\ d$
⟨*proof*⟩

lemma *m-traversal-paths-with-witness-set-containment* :
assumes $q \in states\ M$
and $path\ M\ q\ p$
and $d \in set\ repSets$
and $Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p)$
and $\bigwedge p'\ p''.$
$p = p' @ p'' \Longrightarrow p'' \neq [] \Longrightarrow$
$\neg (\exists\ d{\in}set\ repSets.$
$Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p'))$

and $\bigwedge q . q \in states\ M \Longrightarrow \exists d \in set\ repSets.\ q \in fst\ d$
and $\bigwedge d . d \in set\ repSets \Longrightarrow snd\ d \subseteq fst\ d$
shows $\exists\ d' . (p,d') \in (m\text{-}traversal\text{-}paths\text{-}with\text{-}witness\ M\ q\ repSets\ m)$
⟨*proof*⟩

lemma *m-traversal-path-exist* :
assumes *completely-specified M*
and $q \in states\ M$
and $inputs\ M \neq \{\}$
and $\bigwedge q . q \in states\ M \Longrightarrow \exists d \in set\ D.\ q \in fst\ d$
and $\bigwedge d . d \in set\ D \Longrightarrow snd\ d \subseteq fst\ d$
shows $\exists\ p'\ d' . (p',d') \in (m\text{-}traversal\text{-}paths\text{-}with\text{-}witness\ M\ q\ D\ m)$
⟨*proof*⟩

lemma *m-traversal-path-extension-exist* :
assumes *completely-specified M*
and $q \in states\ M$
and $inputs\ M \neq \{\}$
and $\bigwedge q . q \in states\ M \Longrightarrow \exists d \in set\ D.\ q \in fst\ d$
and $\bigwedge d . d \in set\ D \Longrightarrow snd\ d \subseteq fst\ d$
and *path M q p1*
and $find\ (\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p1))\ D = None$
shows $\exists\ p2\ d' . (p1@p2,d') \in (m\text{-}traversal\text{-}paths\text{-}with\text{-}witness\ M\ q\ D\ m)$
⟨*proof*⟩

lemma *m-traversal-path-extension-exist-for-transition* :
assumes *completely-specified M*
and $q \in states\ M$
and $inputs\ M \neq \{\}$
and $\bigwedge q . q \in states\ M \Longrightarrow \exists d \in set\ D.\ q \in fst\ d$
and $\bigwedge d . d \in set\ D \Longrightarrow snd\ d \subseteq fst\ d$
and *path M q p1*
and $find\ (\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p1))\ D = None$
and $t \in transitions\ M$
and $t\text{-}source\ t = target\ q\ p1$
shows $\exists\ p2\ d' . (p1@[t]@p2,d') \in (m\text{-}traversal\text{-}paths\text{-}with\text{-}witness\ M\ q\ D\ m)$
⟨*proof*⟩

end

41 Test Suites

This theory introduces a predicate *implies-completeness* and proves that any test suite satisfying this predicate is sufficient to check the reduction conformance relation between two (possibly nondeterministic FSMs)

theory *Test-Suite*
imports *Helper-Algorithms Adaptive-Test-Case Traversal-Set*
begin

41.1 Preliminary Definitions

type-synonym ('*a*,'*b*,'*c*) *preamble* = ('*a*,'*b*,'*c*) *fsm*
type-synonym ('*a*,'*b*,'*c*) *traversal-path* = ('*a* × '*b* × '*c* × '*a*) *list*
type-synonym ('*a*,'*b*,'*c*) *separator* = ('*a*,'*b*,'*c*) *fsm*

A test suite contains of 1) a set of d-reachable states with their associated preambles 2) a map from d-reachable states to their associated m-traversal paths 3) a map from d-reachable states and associated m-traversal paths to the set of states to r-distinguish the targets of those paths from 4) a map from pairs of r-distinguishable states to a separator

datatype ('*a*,'*b*,'*c*,'*d*) *test-suite* = *Test-Suite* ('*a* × ('*a*,'*b*,'*c*) *preamble*) *set*
'*a* ⇒ ('*a*,'*b*,'*c*) *traversal-path set*
('*a* × ('*a*,'*b*,'*c*) *traversal-path*) ⇒ '*a set*
('*a* × '*a*) ⇒ (('*d*,'*b*,'*c*) *separator* × '*d* × '*d*) *set*

41.2 A Sufficiency Criterion for Reduction Testing

fun *implies-completeness-for-repetition-sets* :: ('*a*,'*b*,'*c*,'*d*) *test-suite* ⇒ ('*a*,'*b*,'*c*) *fsm* ⇒ *nat* ⇒ ('*a set* × '*a set*) *list* ⇒ *bool* **where**
implies-completeness-for-repetition-sets (*Test-Suite prs tps rd-targets separators*) *M m repetition-sets* =
((*initial M*,*initial-preamble M*) ∈ *prs*
∧ (∀ *q P* . (*q*,*P*) ∈ *prs* ⟶ (*is-preamble P M q*) ∧ (*tps q*) ≠ {})
∧ (∀ *q1 q2 A d1 d2* . (*A*,*d1*,*d2*) ∈ *separators* (*q1*,*q2*) ⟶ (*A*,*d2*,*d1*) ∈ *separators* (*q2*,*q1*) ∧ *is-separator M q1 q2 A d1 d2*)
∧ (∀ *q* . *q* ∈ *states M* ⟶ (∃ *d* ∈ *set repetition-sets*. *q* ∈ *fst d*))
∧ (∀ *d* . *d* ∈ *set repetition-sets* ⟶ ((*fst d* ⊆ *states M*) ∧ (*snd d* = *fst d* ∩ *fst* ‘ *prs*) ∧ (∀ *q1 q2* . *q1* ∈ *fst d* ⟶ *q2* ∈ *fst d* ⟶ *q1* ≠ *q2* ⟶ *separators* (*q1*,*q2*) ≠ {})))
∧ (∀ *q* . *q* ∈ *image fst prs* ⟶ *tps q* ⊆ {*p1* . ∃ *p2 d* . (*p1*@*p2*,*d*) ∈ *m-traversal-paths-with-witness M q repetition-sets m*} ∧ *fst* ‘ (*m-traversal-paths-with-witness M q repetition-sets m*) ⊆ *tps q*)
∧ (∀ *q p d* . *q* ∈ *image fst prs* ⟶ (*p*,*d*) ∈ *m-traversal-paths-with-witness M q repetition-sets m* ⟶
((∀ *p1 p2 p3* . *p*=*p1*@*p2*@*p3* ⟶ *p2* ≠ [] ⟶ *target q p1* ∈ *fst d* ⟶ *target q* (*p1*@*p2*) ∈ *fst d* ⟶ *target q p1* ≠ *target q* (*p1*@*p2*) ⟶ (*p1* ∈ *tps q* ∧ (*p1*@*p2*) ∈ *tps q* ∧ *target q p1* ∈ *rd-targets* (*q*,(*p1*@*p2*)) ∧ *target q* (*p1*@*p2*) ∈ *rd-targets* (*q*,*p1*)))

$\wedge$ ($\forall$ *p1 p2 q'* . *p=p1@p2* $\longrightarrow$ *q'* $\in$ *image fst prs* $\longrightarrow$ *target q p1* $\in$ *fst d* $\longrightarrow$ *q'* $\in$ *fst d* $\longrightarrow$ *target q p1* $\neq$ *q'* $\longrightarrow$ (*p1* $\in$ *tps q* $\wedge$ [] $\in$ *tps q'* $\wedge$ *target q p1* $\in$ *rd-targets* (*q'*,[]) $\wedge$ *q'* $\in$ *rd-targets* (*q*,*p1*))))
$\wedge$ ($\forall$ *q1 q2* . *q1* $\neq$ *q2* $\longrightarrow$ *q1* $\in$ *snd d* $\longrightarrow$ *q2* $\in$ *snd d* $\longrightarrow$ ([] $\in$ *tps q1* $\wedge$ [] $\in$ *tps q2* $\wedge$ *q1* $\in$ *rd-targets* (*q2*,[]) $\wedge$ *q2* $\in$ *rd-targets* (*q1*,[]))))
)

definition *implies-completeness* :: (*'a,'b,'c,'d*) *test-suite* $\Rightarrow$ (*'a,'b,'c*) *fsm* $\Rightarrow$ *nat* $\Rightarrow$ *bool* **where**
implies-completeness T M m = ($\exists$ *repetition-sets* . *implies-completeness-for-repetition-sets T M m repetition-sets*)

lemma *implies-completeness-for-repetition-sets-simps* :
assumes *implies-completeness-for-repetition-sets* (*Test-Suite prs tps rd-targets separators*) *M m repetition-sets*
shows (*initial M*,*initial-preamble M*) $\in$ *prs*
and $\bigwedge$ *q P* . (*q*,*P*) $\in$ *prs* $\Longrightarrow$ (*is-preamble P M q*) $\wedge$ (*tps q*) $\neq$ {}
and $\bigwedge$ *q1 q2 A d1 d2* . (*A*,*d1*,*d2*) $\in$ *separators* (*q1*,*q2*) $\Longrightarrow$ (*A*,*d2*,*d1*) $\in$ *separators* (*q2*,*q1*) $\wedge$ *is-separator M q1 q2 A d1 d2*
and $\bigwedge$ *q* . *q* $\in$ *states M* $\Longrightarrow$ ($\exists$ *d* $\in$ *set repetition-sets*. *q* $\in$ *fst d*)
and $\bigwedge$ *d* . *d* $\in$ *set repetition-sets* $\Longrightarrow$ (*fst d* $\subseteq$ *states M*) $\wedge$ (*snd d* = *fst d* $\cap$ *fst* ' *prs*)
and $\bigwedge$ *d q1 q2* . *d* $\in$ *set repetition-sets* $\Longrightarrow$ *q1* $\in$ *fst d* $\Longrightarrow$ *q2* $\in$ *fst d* $\Longrightarrow$ *q1* $\neq$ *q2* $\Longrightarrow$ *separators* (*q1*,*q2*) $\neq$ {}
and $\bigwedge$ *q* . *q* $\in$ *image fst prs* $\Longrightarrow$ *tps q* $\subseteq$ {*p1* . $\exists$ *p2 d* . (*p1@p2*,*d*) $\in$ *m-traversal-paths-with-witness M q repetition-sets m*} $\wedge$ *fst* ' (*m-traversal-paths-with-witness M q repetition-sets m*) $\subseteq$ *tps q*
and $\bigwedge$ *q p d p1 p2 p3* . *q* $\in$ *image fst prs* $\Longrightarrow$ (*p*,*d*) $\in$ *m-traversal-paths-with-witness M q repetition-sets m* $\Longrightarrow$ *p=p1@p2@p3* $\Longrightarrow$ *p2* $\neq$ [] $\Longrightarrow$ *target q p1* $\in$ *fst d* $\Longrightarrow$ *target q* (*p1@p2*) $\in$ *fst d* $\Longrightarrow$ *target q p1* $\neq$ *target q* (*p1@p2*) $\Longrightarrow$ (*p1* $\in$ *tps q* $\wedge$ (*p1@p2*) $\in$ *tps q* $\wedge$ *target q p1* $\in$ *rd-targets* (*q*,(*p1@p2*)) $\wedge$ *target q* (*p1@p2*) $\in$ *rd-targets* (*q*,*p1*))
and $\bigwedge$ *q p d p1 p2 q'* . *q* $\in$ *image fst prs* $\Longrightarrow$ (*p*,*d*) $\in$ *m-traversal-paths-with-witness M q repetition-sets m* $\Longrightarrow$ *p=p1@p2* $\Longrightarrow$ *q'* $\in$ *image fst prs* $\Longrightarrow$ *target q p1* $\in$ *fst d* $\Longrightarrow$ *q'* $\in$ *fst d* $\Longrightarrow$ *target q p1* $\neq$ *q'* $\Longrightarrow$ (*p1* $\in$ *tps q* $\wedge$ [] $\in$ *tps q'* $\wedge$ *target q p1* $\in$ *rd-targets* (*q'*,[]) $\wedge$ *q'* $\in$ *rd-targets* (*q*,*p1*))
and $\bigwedge$ *q p d q1 q2* . *q* $\in$ *image fst prs* $\Longrightarrow$ (*p*,*d*) $\in$ *m-traversal-paths-with-witness M q repetition-sets m* $\Longrightarrow$ *q1* $\neq$ *q2* $\Longrightarrow$ *q1* $\in$ *snd d* $\Longrightarrow$ *q2* $\in$ *snd d* $\Longrightarrow$ ([] $\in$ *tps q1* $\wedge$ [] $\in$ *tps q2* $\wedge$ *q1* $\in$ *rd-targets* (*q2*,[]) $\wedge$ *q2* $\in$ *rd-targets* (*q1*,[]))
⟨*proof*⟩

41.3 A Pass Relation for Test Suites and Reduction Testing

fun *passes-test-suite* :: (*'a,'b,'c*) *fsm* $\Rightarrow$ (*'a,'b,'c,'d*) *test-suite* $\Rightarrow$ (*'e,'b,'c*) *fsm* $\Rightarrow$ *bool* **where**
passes-test-suite M (*Test-Suite prs tps rd-targets separators*) *M'* = (
— Reduction on preambles: as the preambles contain all responses of M to their

chosen inputs, M' must not exhibit any other response

($\forall$ *q P io x y y'* . (*q*,*P*) $\in$ *prs* $\longrightarrow$ *io*@[(*x*,*y*)] $\in$ *L P* $\longrightarrow$ *io*@[(*x*,*y'*)] $\in$ *L M'* $\longrightarrow$ *io*@[(*x*,*y'*)] $\in$ *L P*)

— Reduction on traversal-paths applied after preambles (i.e., completed paths in preambles) - note that tps q is not necessarily prefix-complete

$\wedge$ ($\forall$ *q P pP ioT pT x y y'* . (*q*,*P*) $\in$ *prs* $\longrightarrow$ *path P* (*initial P*) *pP* $\longrightarrow$ *target* (*initial P*) *pP* = *q* $\longrightarrow$ *pT* $\in$ *tps q* $\longrightarrow$ *ioT*@[(*x*,*y*)] $\in$ *set* (*prefixes* (*p-io pT*)) $\longrightarrow$ (*p-io pP*)@*ioT*@[(*x*,*y'*)] $\in$ *L M'* $\longrightarrow$ ($\exists$ *pT'* . *pT'* $\in$ *tps q* $\wedge$ *ioT*@[(*x*,*y'*)] $\in$ *set* (*prefixes* (*p-io pT'*))))

— Passing separators: if M' contains an IO-sequence that in the test suite leads through a preamble and an m-traversal path and the target of the latter is to be r-distinguished from some other state, then M' passes the corresponding ATC

$\wedge$ ($\forall$ *q P pP pT* . (*q*,*P*) $\in$ *prs* $\longrightarrow$ *path P* (*initial P*) *pP* $\longrightarrow$ *target* (*initial P*) *pP* = *q* $\longrightarrow$ *pT* $\in$ *tps q* $\longrightarrow$ (*p-io pP*)@(*p-io pT*) $\in$ *L M'* $\longrightarrow$ ($\forall$ *q' A d1 d2 qT* . *q'* $\in$ *rd-targets* (*q*,*pT*) $\longrightarrow$ (*A*,*d1*,*d2*) $\in$ *separators* (*target q pT*, *q'*) $\longrightarrow$ *qT* $\in$ *io-targets M'* ((*p-io pP*)@(*p-io pT*)) (*initial M'*) $\longrightarrow$ *pass-separator-ATC M' A qT d2*))

)

41.4 Soundness of Sufficient Test Suites

lemma *passes-test-suite-soundness-helper-1* :
 assumes *is-preamble P M q*
 and *observable M*
 and *io*@[(*x*,*y*)] $\in$ *L P*
 and *io*@[(*x*,*y'*)] $\in$ *L M*
shows *io*@[(*x*,*y'*)] $\in$ *L P*
⟨*proof*⟩

lemma *passes-test-suite-soundness* :
 assumes *implies-completeness* (*Test-Suite prs tps rd-targets separators*) *M m*
 and *observable M*
 and *observable M'*
 and *inputs M'* = *inputs M*
 and *completely-specified M*
 and *L M'* $\subseteq$ *L M*
shows *passes-test-suite M* (*Test-Suite prs tps rd-targets separators*) *M'*
⟨*proof*⟩

41.5 Exhaustiveness of Sufficient Test Suites

This subsection shows that test suites satisfying the sufficiency criterion are exhaustive. That is, for a System Under Test with at most m states that contains an error (i.e.: is not a reduction) a test suite sufficient for m will not pass.

41.5.1 R Functions

definition R :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $'a \Rightarrow 'a \Rightarrow ('a \times 'b \times 'c \times 'a)$ *list* $\Rightarrow ('a \times 'b \times 'c \times 'a)$ *list* $\Rightarrow ('a \times 'b \times 'c \times 'a)$ *list set* **where**
$R\ M\ q\ q'\ pP\ p = \{pP\ @\ p' \mid p'\ .\ p' \neq [] \wedge target\ q\ p' = q' \wedge (\exists\ p''\ .\ p = p'@p'')\}$

definition RP :: $('a,'b,'c)$ *fsm* $\Rightarrow 'a \Rightarrow 'a \Rightarrow ('a \times 'b \times 'c \times 'a)$ *list* $\Rightarrow ('a \times 'b \times 'c \times 'a)$ *list* $\Rightarrow ('a \times ('a,'b,'c)$ *preamble*$)$ *set* $\Rightarrow ('d,'b,'c)$ *fsm* $\Rightarrow ('a \times 'b \times 'c \times 'a)$ *list set* **where**
$RP\ M\ q\ q'\ pP\ p\ PS\ M' = (if\ \exists\ P'\ .\ (q',P') \in PS\ then\ insert\ (SOME\ pP'\ .\ \exists\ P'\ .\ (q',P') \in PS \wedge path\ P'\ (initial\ P')\ pP' \wedge target\ (initial\ P')\ pP' = q' \wedge p\text{-}io\ pP' \in L\ M')\ (R\ M\ q\ q'\ pP\ p)\ else\ (R\ M\ q\ q'\ pP\ p))$

lemma *RP-from-R* :
assumes $\bigwedge\ q\ P\ .\ (q,P) \in PS \Longrightarrow is\text{-}preamble\ P\ M\ q$
and $\bigwedge\ q\ P\ io\ x\ y\ y'\ .\ (q,P) \in PS \Longrightarrow io@[(x,y)] \in L\ P \Longrightarrow io@[(x,y')] \in L\ M' \Longrightarrow io@[(x,y')] \in L\ P$
and $completely\text{-}specified\ M'$
and $inputs\ M' = inputs\ M$
shows $(RP\ M\ q\ q'\ pP\ p\ PS\ M' = R\ M\ q\ q'\ pP\ p)$
$\vee (\exists\ P'\ pP'\ .\ (q',P') \in PS \wedge$
$path\ P'\ (initial\ P')\ pP' \wedge$
$target\ (initial\ P')\ pP' = q' \wedge$
$path\ M\ (initial\ M)\ pP' \wedge$
$target\ (initial\ M)\ pP' = q' \wedge$
$p\text{-}io\ pP' \in L\ M' \wedge$
$RP\ M\ q\ q'\ pP\ p\ PS\ M' =$
$insert\ pP'\ (R\ M\ q\ q'\ pP\ p))$
⟨*proof*⟩

lemma *RP-from-R-inserted* :
assumes $\bigwedge\ q\ P\ .\ (q,P) \in PS \Longrightarrow is\text{-}preamble\ P\ M\ q$
and $\bigwedge\ q\ P\ io\ x\ y\ y'\ .\ (q,P) \in PS \Longrightarrow io@[(x,y)] \in L\ P \Longrightarrow io@[(x,y')] \in L\ M' \Longrightarrow io@[(x,y')] \in L\ P$
and $completely\text{-}specified\ M'$
and $inputs\ M' = inputs\ M$
and $pP' \in RP\ M\ q\ q'\ pP\ p\ PS\ M'$
and $pP' \notin R\ M\ q\ q'\ pP\ p$
obtains P' **where** $(q',P') \in PS$
$path\ P'\ (initial\ P')\ pP'$
$target\ (initial\ P')\ pP' = q'$
$path\ M\ (initial\ M)\ pP'$
$target\ (initial\ M)\ pP' = q'$
$p\text{-}io\ pP' \in L\ M'$
$RP\ M\ q\ q'\ pP\ p\ PS\ M' = insert\ pP'\ (R\ M\ q\ q'\ pP\ p)$

⟨*proof*⟩

lemma *finite-R* :
assumes *path M q p*
shows *finite* (*R M q q′ pP p*)
⟨*proof*⟩

lemma *finite-RP* :
assumes *path M q p*
and $\bigwedge$ *q P* . (*q*,*P*) ∈ *PS* ⟹ *is-preamble P M q*
and $\bigwedge$ *q P io x y y′* . (*q*,*P*) ∈ *PS* ⟹ *io*@[(*x*,*y*)] ∈ *L P* ⟹ *io*@[(*x*,*y′*)] ∈ *L M′* ⟹ *io*@[(*x*,*y′*)] ∈ *L P*
and *completely-specified M′*
and *inputs M′* = *inputs M*
shows *finite* (*RP M q q′ pP p PS M′*)
⟨*proof*⟩

lemma *R-component-ob* :
assumes *pR′* ∈ *R M q q′ pP p*
obtains *pR* **where** *pR′* = *pP*@*pR*
⟨*proof*⟩

lemma *R-component* :
assumes (*pP*@*pR*) ∈ *R M q q′ pP p*
shows *pR* = *take* (*length pR*) *p*
and *length pR* ≤ *length p*
and *t-target* (*p* ! (*length pR* − *1*)) = *q′*
and *pR* ≠ []
⟨*proof*⟩

lemma *R-component-observable* :
assumes *pP*@*pR* ∈ *R M* (*target* (*initial M*) *pP*) *q′ pP p*
and *observable M*
and *path M* (*initial M*) *pP*
and *path M* (*target* (*initial M*) *pP*) *p*
shows *io-targets M* (*p-io pP* @ *p-io pR*) (*initial M*) = {*target* (*target* (*initial M*) *pP*) (*take* (*length pR*) *p*)}
⟨*proof*⟩

lemma *R-count* :
assumes *minimal-sequence-to-failure-extending-preamble-path M M′ PS pP io*
and *observable M*
and *observable M′*
and $\bigwedge$ *q P*. (*q*, *P*) ∈ *PS* ⟹ *is-preamble P M q*

and *path M* (*target* (*initial M*) *pP*) *p*
and *butlast io* = *p-io p* @ *ioX*
shows *card* (⋃ (*image* (λ *pR* . *io-targets M′* (*p-io pR*) (*initial M′*)) (*R M* (*target* (*initial M*) *pP*) *q′ pP p*))) = *card* (*R M* (*target* (*initial M*) *pP*) *q′ pP p*)
(**is** *card ?Tgts* = *card ?R*)
and ⋀ *pR* . *pR* ∈ (*R M* (*target* (*initial M*) *pP*) *q′ pP p*) ⟹ ∃ *q* . *io-targets M′* (*p-io pR*) (*initial M′*) = {*q*}
and ⋀ *pR1 pR2* . *pR1* ∈ (*R M* (*target* (*initial M*) *pP*) *q′ pP p*) ⟹
pR2 ∈ (*R M* (*target* (*initial M*) *pP*) *q′ pP p*) ⟹
pR1 ≠ *pR2* ⟹
io-targets M′ (*p-io pR1*) (*initial M′*) ∩ *io-targets M′* (*p-io pR2*) (*initial M′*) = {}
⟨*proof*⟩

lemma *R-update* :
R M q q′ pP (*p*@[*t*]) = (*if* (*target q* (*p*@[*t*]) = *q′*)
then insert (*pP*@*p*@[*t*]) (*R M q q′ pP p*)
else (*R M q q′ pP p*))
(**is** *?R1* = *?R2*)
⟨*proof*⟩

lemma *R-union-card-is-suffix-length* :
assumes *path M* (*initial M*) *pP*
and *path M* (*target* (*initial M*) *pP*) *p*
shows (∑ *q* ∈ *states M* . *card* (*R M* (*target* (*initial M*) *pP*) *q pP p*)) = *length p*
⟨*proof*⟩

lemma *RP-count* :
assumes *minimal-sequence-to-failure-extending-preamble-path M M′ PS pP io*
and *observable M*
and *observable M′*
and ⋀ *q P*. (*q*, *P*) ∈ *PS* ⟹ *is-preamble P M q*
and *path M* (*target* (*initial M*) *pP*) *p*
and *butlast io* = *p-io p* @ *ioX*
and ⋀ *q P io x y y′* . (*q*,*P*) ∈ *PS* ⟹ *io*@[(*x*,*y*)] ∈ *L P* ⟹ *io*@[(*x*,*y′*)] ∈ *L M′* ⟹ *io*@[(*x*,*y′*)] ∈ *L P*
and *completely-specified M′*
and *inputs M′* = *inputs M*
shows *card* (⋃ (*image* (λ *pR* . *io-targets M′* (*p-io pR*) (*initial M′*)) (*RP M* (*target* (*initial M*) *pP*) *q′ pP p PS M′*)))
= *card* (*RP M* (*target* (*initial M*) *pP*) *q′ pP p PS M′*)
(**is** *card ?Tgts* = *card ?RP*)
and ⋀ *pR* . *pR* ∈ (*RP M* (*target* (*initial M*) *pP*) *q′ pP p PS M′*) ⟹ ∃ *q* . *io-targets M′* (*p-io pR*) (*initial M′*) = {*q*}
and ⋀ *pR1 pR2* . *pR1* ∈ (*RP M* (*target* (*initial M*) *pP*) *q′ pP p PS M′*) ⟹ *pR2*

$\in$ (*RP M* (*target* (*initial M*) *pP*) q' *pP p PS* M') $\Longrightarrow$ *pR1* $\neq$ *pR2* $\Longrightarrow$ *io-targets* M' (*p-io pR1*) (*initial* M') $\cap$ *io-targets* M' (*p-io pR2*) (*initial* M') = {}
⟨*proof*⟩

lemma *RP-target*:
assumes *pR* $\in$ (*RP M q* q' *pP p PS* M')
assumes $\bigwedge$ *q P* . (*q*,*P*) $\in$ *PS* $\Longrightarrow$ *is-preamble P M q*
and $\bigwedge$ *q P io x y* y' . (*q*,*P*) $\in$ *PS* $\Longrightarrow$ *io*@[(*x*,*y*)] $\in$ *L P* $\Longrightarrow$ *io*@[(*x*,y')] $\in$ *L* M' $\Longrightarrow$ *io*@[(*x*,y')] $\in$ *L P*
and *completely-specified* M'
and *inputs* M' = *inputs M*
shows *target* (*initial M*) *pR* = q'
⟨*proof*⟩

41.5.2 Proof of Exhaustiveness

lemma *passes-test-suite-exhaustiveness-helper-1* :
assumes *completely-specified* M'
and *inputs* M' = *inputs M*
and *observable M*
and *observable* M'
and (*q*,*P*) $\in$ *PS*
and *path P* (*initial P*) *pP*
and *target* (*initial P*) *pP* = *q*
and *p-io pP* @ *p-io p* $\in$ *L* M'
and (*p*, *d*) $\in$ *m-traversal-paths-with-witness M q repetition-sets m*
and *implies-completeness-for-repetition-sets* (*Test-Suite PS tps rd-targets separators*) *M m repetition-sets*
and *passes-test-suite M* (*Test-Suite PS tps rd-targets separators*) M'
and $q' \neq q''$
and q' $\in$ *fst d*
and q'' $\in$ *fst d*
and *pR1* $\in$ (*RP M q* q' *pP p PS* M')
and *pR2* $\in$ (*RP M q* q'' *pP p PS* M')
shows *io-targets* M' (*p-io pR1*) (*initial* M') $\cap$ *io-targets* M' (*p-io pR2*) (*initial* M') = {}
⟨*proof*⟩

lemma *passes-test-suite-exhaustiveness* :
assumes *passes-test-suite M* (*Test-Suite prs tps rd-targets separators*) M'
and *implies-completeness* (*Test-Suite prs tps rd-targets separators*) *M m*
and *observable M*
and *observable* M'
and *inputs* M' = *inputs M*

and *inputs* $M \neq \{\}$
and *completely-specified* M
and *completely-specified* M'
and *size* $M' \leq m$
shows $L\ M' \subseteq L\ M$
⟨*proof*⟩

41.6 Completeness of Sufficient Test Suites

This subsection combines the soundness and exhaustiveness properties of sufficient test suites to show completeness: for any System Under Test with at most m states a test suite sufficient for m passes if and only if the System Under Test is a reduction of the specification.

lemma *passes-test-suite-completeness* :
assumes *implies-completeness* $T\ M\ m$
and *observable* M
and *observable* M'
and *inputs* M' = *inputs* M
and *inputs* $M \neq \{\}$
and *completely-specified* M
and *completely-specified* M'
and *size* $M' \leq m$
shows $(L\ M' \subseteq L\ M) \longleftrightarrow$ *passes-test-suite* $M\ T\ M'$
⟨*proof*⟩

41.7 Additional Test Suite Properties

fun *is-finite-test-suite* :: ('a,'b,'c,'d) *test-suite* ⇒ *bool* **where**
is-finite-test-suite (*Test-Suite prs tps rd-targets separators*) =
((*finite prs*) ∧ (∀ *q p* . *q* ∈ *fst* ' *prs* ⟶ *finite* (*rd-targets* (*q*,*p*))) ∧ (∀ *q q'* . *finite* (*separators* (*q*,*q'*))))

end

42 Representing Test Suites as Sets of Input-Output Sequences

This theory describes the representation of test suites as sets of input-output sequences and defines a pass relation for this representation.

theory *Test-Suite-IO*
imports *Test-Suite Maximal-Path-Trie*
begin

fun *test-suite-to-io* :: ('a,'b,'c) *fsm* ⇒ ('a,'b,'c,'d) *test-suite* ⇒ ('b × 'c) *list set*
where

test-suite-to-io M (*Test-Suite prs tps rd-targets atcs*) =
($\bigcup$ (*q*,*P*) ∈ *prs* . *L P*)
∪ ($\bigcup${(λ *io′* . *p-io p* @ *io′*) ‘ (*set* (*prefixes* (*p-io pt*))) | *p pt* . ∃ *q P* . (*q*,*P*) ∈ *prs* ∧ *path P* (*initial P*) *p* ∧ *target* (*initial P*) *p* = *q* ∧ *pt* ∈ *tps q*})
∪ ($\bigcup${(λ *io-atc* . *p-io p* @ *p-io pt* @ *io-atc*) ‘ (*atc-to-io-set* (*from-FSM M* (*target q pt*)) *A*) | *p pt q A* . ∃ *P q′ t1 t2* . (*q*,*P*) ∈ *prs* ∧ *path P* (*initial P*) *p* ∧ *target* (*initial P*) *p* = *q* ∧ *pt* ∈ *tps q* ∧ *q′* ∈ *rd-targets* (*q*,*pt*) ∧ (*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*) })

lemma *test-suite-to-io-language* :
assumes *implies-completeness T M m*
shows (*test-suite-to-io M T*) ⊆ *L M*
⟨*proof*⟩

lemma *minimal-io-seq-to-failure* :
assumes ¬ (*L M′* ⊆ *L M*)
and *inputs M′* = *inputs M*
and *completely-specified M*
obtains *io x y y′* **where** *io*@[(*x*,*y*)] ∈ *L M* **and** *io*@[(*x*,*y′*)] ∉ *L M* **and** *io*@[(*x*,*y′*)] ∈ *L M′*
⟨*proof*⟩

lemma *observable-minimal-path-to-failure* :
assumes ¬ (*L M′* ⊆ *L M*)
and *observable M*
and *observable M′*
and *inputs M′* = *inputs M*
and *completely-specified M*
and *completely-specified M′*
obtains *p p′ t t′* **where** *path M* (*initial M*) (*p*@[*t*])
and *path M′* (*initial M′*) (*p′*@[*t′*])
and *p-io p′* = *p-io p*
and *t-input t′* = *t-input t*
and ¬(∃ *t″* . *t″* ∈ *transitions M* ∧ *t-source t″* = *target* (*initial M*) *p* ∧ *t-input t″* = *t-input t* ∧ *t-output t″* = *t-output t′*)
⟨*proof*⟩

lemma *test-suite-to-io-pass* :
assumes *implies-completeness T M m*
and *observable M*
and *observable M′*
and *inputs M′* = *inputs M*
and *inputs M* ≠ {}

and *completely-specified M*
and *completely-specified M′*
shows *pass-io-set M′* (*test-suite-to-io M T*) = *passes-test-suite M T M′*
⟨*proof*⟩

lemma *test-suite-to-io-finite* :
assumes *implies-completeness T M m*
and *is-finite-test-suite T*
shows *finite* (*test-suite-to-io M T*)
⟨*proof*⟩

42.1 Calculating the Sets of Sequences

abbreviation *L-acyclic M* ≡ *LS-acyclic M* (*initial M*)

fun *test-suite-to-io′* :: (′*a*,′*b*,′*c*) *fsm* ⇒ (′*a*,′*b*,′*c*,′*d*) *test-suite* ⇒ (′*b* × ′*c*) *list set*
where
test-suite-to-io′ M (*Test-Suite prs tps rd-targets atcs*)
= (⋃ (*q*,*P*) ∈ *prs* .
L-acyclic P
∪ (⋃ *ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*) .
⋃ *pt* ∈ *tps q* .
((λ *io′* . *ioP* @ *io′*) ‘ (*set* (*prefixes* (*p-io pt*))))
∪ (⋃ *q′* ∈ *rd-targets* (*q*,*pt*) .
⋃ (*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*) .
(λ *io-atc* . *ioP* @ *p-io pt* @ *io-atc*) ‘ (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*))))

lemma *test-suite-to-io-code* :
assumes *implies-completeness T M m*
and *is-finite-test-suite T*
and *observable M*
shows *test-suite-to-io M T* = *test-suite-to-io′ M T*
⟨*proof*⟩

42.2 Using Maximal Sequences Only

fun *test-suite-to-io-maximal* :: (′*a*::*linorder*,′*b*::*linorder*,′*c*) *fsm* ⇒ (′*a*,′*b*,′*c*,′*d*::*linorder*) *test-suite* ⇒ (′*b* × ′*c*) *list set* **where**
test-suite-to-io-maximal M (*Test-Suite prs tps rd-targets atcs*) =
remove-proper-prefixes (⋃ (*q*,*P*) ∈ *prs* . *L-acyclic P* ∪ (⋃ *ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*) . ⋃ *pt* ∈ *tps q* . *Set.insert* (*ioP* @ *p-io pt*) (⋃ *q′* ∈ *rd-targets* (*q*,*pt*) . ⋃ (*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*) . (λ *io-atc* . *ioP* @ *p-io pt* @ *io-atc*) ‘ (*remove-proper-prefixes* (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*)))))

lemma *test-suite-to-io-maximal-code* :
assumes *implies-completeness T M m*
and *is-finite-test-suite T*
and *observable M*
shows $\{io' \in (\textit{test-suite-to-io } M\ T)\ .\ \neg\ (\exists\ io''\ .\ io'' \neq [] \wedge io'@io'' \in (\textit{test-suite-to-io } M\ T))\} = \textit{test-suite-to-io-maximal } M\ T$
⟨*proof*⟩

lemma *test-suite-to-io-pass-maximal* :
assumes *implies-completeness T M m*
and *is-finite-test-suite T*
shows $\textit{pass-io-set } M'\ (\textit{test-suite-to-io } M\ T) = \textit{pass-io-set-maximal } M'\ \{io' \in (\textit{test-suite-to-io } M\ T)\ .\ \neg\ (\exists\ io''\ .\ io'' \neq [] \wedge io'@io'' \in (\textit{test-suite-to-io } M\ T))\}$
⟨*proof*⟩

lemma *passes-test-suite-as-maximal-sequences-completeness* :
assumes *implies-completeness T M m*
and *is-finite-test-suite T*
and *observable M*
and *observable M'*
and $\textit{inputs } M' = \textit{inputs } M$
and $\textit{inputs } M \neq \{\}$
and *completely-specified M*
and *completely-specified M'*
and $\textit{size } M' \leq m$
shows $(L\ M' \subseteq L\ M) \longleftrightarrow \textit{pass-io-set-maximal } M'\ (\textit{test-suite-to-io-maximal } M\ T)$
⟨*proof*⟩

lemma *test-suite-to-io-maximal-finite* :
assumes *implies-completeness T M m*
and *is-finite-test-suite T*
and *observable M*
shows *finite* (*test-suite-to-io-maximal M T*)
⟨*proof*⟩

end

43 Calculating Sufficient Test Suites

This theory describes algorithms to calculate test suites that satisfy the sufficiency criterion for a given specification FSM and upper bound m on the number of states in the System Under Test.

theory *Test-Suite-Calculation*
imports *Test-Suite-IO*
begin

43.1 Calculating Path Prefixes that are to be Extended With Adaptive Cest Cases

43.1.1 Calculating Tests along m-Traversal-Paths

fun *prefix-pair-tests* :: $'a \Rightarrow (('a,'b,'c)\ traversal\text{-}path \times ('a\ set \times 'a\ set))\ set \Rightarrow ('a \times ('a,'b,'c)\ traversal\text{-}path \times 'a)\ set$ **where**
prefix-pair-tests q pds
$= \bigcup\{\{(q,p1,(target\ q\ p2)),\ (q,p2,(target\ q\ p1))\} \mid p1\ p2\ .$
$\exists\ (p,(rd,dr)) \in pds\ .$
$(p1,p2) \in set\ (prefix\text{-}pairs\ p) \wedge$
$(target\ q\ p1) \in rd \wedge$
$(target\ q\ p2) \in rd \wedge$
$(target\ q\ p1) \neq (target\ q\ p2)\}$

lemma *prefix-pair-tests-code*[*code*] :
$prefix\text{-}pair\text{-}tests\ q\ pds = (\bigcup(image\ (\lambda\ (p,(rd,dr))\ .\ \bigcup\ (set\ (map\ (\lambda\ (p1,p2)\ .\ \{(q,p1,(target\ q\ p2)),\ (q,p2,(target\ q\ p1))\})\ (filter\ (\lambda\ (p1,p2)\ .\ (target\ q\ p1) \in rd \wedge (target\ q\ p2) \in rd \wedge (target\ q\ p1) \neq (target\ q\ p2))\ (prefix\text{-}pairs\ p)))))\ pds))$
$\langle proof \rangle$

43.1.2 Calculating Tests between Preambles

fun *preamble-prefix-tests'* :: $'a \Rightarrow (('a,'b,'c)\ traversal\text{-}path \times ('a\ set \times 'a\ set))\ list \Rightarrow 'a\ list \Rightarrow ('a \times ('a,'b,'c)\ traversal\text{-}path \times 'a)\ list$ **where**
preamble-prefix-tests' q pds drs =
$concat\ (map\ (\lambda((p,(rd,dr)),q2,p1)\ .\ [(q,p1,q2),\ (q2,[],(target\ q\ p1))])$
$(filter\ (\lambda((p,(rd,dr)),q2,p1)\ .\ (target\ q\ p1) \in rd \wedge q2 \in rd \wedge (target\ q\ p1) \neq q2)$
$(concat\ (map\ (\lambda((p,(rd,dr)),q2)\ .\ map\ (\lambda p1\ .\ ((p,(rd,dr)),q2,p1))\ (prefixes\ p))\ (List.product\ pds\ drs)))))$

definition *preamble-prefix-tests* :: $'a \Rightarrow (('a,'b,'c)\ traversal\text{-}path \times ('a\ set \times 'a\ set))\ set \Rightarrow 'a\ set \Rightarrow ('a \times ('a,'b,'c)\ traversal\text{-}path \times 'a)\ set$ **where**
$preamble\text{-}prefix\text{-}tests\ q\ pds\ drs = \bigcup\{\{(q,p1,q2),\ (q2,[],(target\ q\ p1))\} \mid p1\ q2\ .\ \exists\ (p,(rd,dr)) \in pds\ .\ q2 \in drs \wedge (\exists\ p2\ .\ p = p1@p2) \wedge (target\ q\ p1) \in rd \wedge q2 \in rd \wedge (target\ q\ p1) \neq q2\}$

lemma *preamble-prefix-tests-code*[*code*] :
$preamble\text{-}prefix\text{-}tests\ q\ pds\ drs = (\bigcup(image\ (\lambda\ (p,(rd,dr))\ .\ \bigcup(image\ (\lambda\ (p1,q2)\ .\ \{(q,p1,q2),\ (q2,[],(target\ q\ p1))\})\ (Set.filter\ (\lambda\ (p1,q2)\ .\ (target\ q\ p1) \in rd \wedge q2 \in rd \wedge (target\ q\ p1) \neq q2)\ ((set\ (prefixes\ p)) \times drs))))\ pds))$
$\langle proof \rangle$

43.1.3 Calculating Tests between m-Traversal-Paths Prefixes and Preambles

fun *preamble-pair-tests* :: *'a set set* ⇒ (*'a* × *'a*) *set* ⇒ (*'a* × (*'a*,*'b*,*'c*) *traversal-path* × *'a*) *set* **where**
preamble-pair-tests drss rds = (⋃ *drs* ∈ *drss* . (λ (*q1*,*q2*) . (*q1*,[],*q2*)) ‘ ((*drs* × *drs*) ∩ *rds*))

43.2 Calculating a Test Suite

definition *calculate-test-paths* ::
(*'a*,*'b*,*'c*) *fsm*
⇒ *nat*
⇒ *'a set*
⇒ (*'a* × *'a*) *set*
⇒ (*'a set* × *'a set*) *list*
⇒ ((*'a* ⇒ (*'a*,*'b*,*'c*) *traversal-path set*) × ((*'a* × (*'a*,*'b*,*'c*) *traversal-path*) ⇒ *'a set*))
where
calculate-test-paths M m d-reachable-states r-distinguishable-pairs repetition-sets =
(*let*
paths-with-witnesses
= (*image* (λ *q* . (*q*,*m-traversal-paths-with-witness M q repetition-sets m*)) *d-reachable-states*);
get-paths
= *m2f* (*set-as-map paths-with-witnesses*);
PrefixPairTests
= ⋃ *q* ∈ *d-reachable-states* . ⋃ *mrsps* ∈ *get-paths q* . *prefix-pair-tests q mrsps*;
PreamblePrefixTests
= ⋃ *q* ∈ *d-reachable-states* . ⋃ *mrsps* ∈ *get-paths q* . *preamble-prefix-tests q mrsps d-reachable-states*;
PreamblePairTests
= *preamble-pair-tests* (⋃ (*q*,*pw*) ∈ *paths-with-witnesses* . ((λ (*p*,(*rd*,*dr*)) . *dr*) ‘ *pw*)) *r-distinguishable-pairs*;
tests
= *PrefixPairTests* ∪ *PreamblePrefixTests* ∪ *PreamblePairTests*;
tps'
= *m2f-by* ⋃ (*set-as-map* (*image* (λ (*q*,*p*) . (*q*, *image fst p*)) *paths-with-witnesses*));
tps''
= *m2f* (*set-as-map* (*image* (λ (*q*,*p*,*q'*) . (*q*,*p*)) *tests*));
tps
= (λ *q* . *tps' q* ∪ *tps'' q*);
rd-targets
= *m2f* (*set-as-map* (*image* (λ (*q*,*p*,*q'*) . ((*q*,*p*),*q'*)) *tests*))
in
(*tps*, *rd-targets*))

definition *combine-test-suite* ::
$('a,'b,'c)$ *fsm*
$\Rightarrow$ *nat*
$\Rightarrow ('a \times ('a,'b,'c)$ *preamble*) *set*
$\Rightarrow (('a \times 'a) \times (('d,'b,'c)$ *separator* $\times 'd \times 'd))$ *set*
$\Rightarrow ('a$ *set* $\times 'a$ *set*) *list*
$\Rightarrow ('a,'b,'c,'d)$ *test-suite*
where
combine-test-suite M m states-with-preambles pairs-with-separators repetition-sets =
(*let drs* = *image fst states-with-preambles*;
rds = *image fst pairs-with-separators*;
tps-and-targets = *calculate-test-paths M m drs rds repetition-sets*;
atcs = *m2f* (*set-as-map pairs-with-separators*)
in (*Test-Suite states-with-preambles* (*fst tps-and-targets*) (*snd tps-and-targets*) *atcs*))

definition *calculate-test-suite-for-repetition-sets* ::
$('a::linorder,'b::linorder,'c)$ *fsm* $\Rightarrow$ *nat* $\Rightarrow ('a$ *set* $\times 'a$ *set*) *list* $\Rightarrow ('a,'b,'c, ('a \times 'a) + 'a)$ *test-suite*
where
calculate-test-suite-for-repetition-sets M m repetition-sets =
(*let*
states-with-preambles = *d-reachable-states-with-preambles M*;
pairs-with-separators = *image* $(\lambda((q1,q2),A)\ .\ ((q1,q2),A,Inr\ q1,Inr\ q2))$ (*r-distinguishable-state-pairs-with-separators M*)
in combine-test-suite M m states-with-preambles pairs-with-separators repetition-sets)

43.3 Sufficiency of the Calculated Test Suite

lemma *calculate-test-suite-for-repetition-sets-sufficient-and-finite* :
fixes *M* :: $('a::linorder,'b::linorder,'c)$ *fsm*
assumes *observable M*
and *completely-specified M*
and *inputs* $M \neq \{\}$
and $\bigwedge q.\ q \in FSM.states\ M \Longrightarrow \exists\, d{\in}set\ RepSets.\ q \in fst\ d$
and $\bigwedge d.\ d \in set\ RepSets \Longrightarrow fst\ d \subseteq states\ M \wedge (snd\ d = fst\ d \cap fst$ ‘ *d-reachable-states-with-preambles M*)
and $\bigwedge q1\ q2\ d.\ d \in set\ RepSets \Longrightarrow q1{\in}fst\ d \Longrightarrow q2{\in}fst\ d \Longrightarrow q1 \neq q2 \Longrightarrow (q1, q2) \in fst$ ‘ *r-distinguishable-state-pairs-with-separators M*
shows *implies-completeness* (*calculate-test-suite-for-repetition-sets M m RepSets*) *M m*
and *is-finite-test-suite* (*calculate-test-suite-for-repetition-sets M m RepSets*)
⟨*proof*⟩

43.4 Two Complete Example Implementations

43.4.1 Naive Repetition Set Strategy

definition *calculate-test-suite-naive* :: ('a::linorder,'b::linorder,'c) fsm ⇒ nat ⇒ ('a,'b,'c, ('a × 'a) + 'a) test-suite **where**
calculate-test-suite-naive M m = calculate-test-suite-for-repetition-sets M m (maximal-repetition-sets-from-sep M)

definition *calculate-test-suite-naive-as-io-sequences* :: ('a::linorder,'b::linorder,'c) fsm ⇒ nat ⇒ ('b × 'c) list set **where**
calculate-test-suite-naive-as-io-sequences M m = test-suite-to-io-maximal M (calculate-test-suite-naive M m)

lemma *calculate-test-suite-naive-completeness* :
fixes *M* :: ('a::linorder,'b::linorder,'c) fsm
assumes *observable M*
and *observable M'*
and *inputs M' = inputs M*
and *inputs M ≠ {}*
and *completely-specified M*
and *completely-specified M'*
and *size M' ≤ m*
shows (L M' ⊆ L M) ⟷ *passes-test-suite M (calculate-test-suite-naive M m) M'*
and (L M' ⊆ L M) ⟷ *pass-io-set-maximal M' (calculate-test-suite-naive-as-io-sequences M m)*
⟨*proof*⟩

definition *calculate-test-suite-naive-as-io-sequences-with-assumption-check* :: ('a::linorder,'b::linorder,'c) fsm ⇒ nat ⇒ String.literal + (('b × 'c) list set) **where**
calculate-test-suite-naive-as-io-sequences-with-assumption-check M m =
(if inputs M ≠ {}
then if observable M
then if completely-specified M
then (Inr (test-suite-to-io-maximal M (calculate-test-suite-naive M m)))
else (Inl (STR ''specification is not completely specified''))
else (Inl (STR ''specification is not observable''))
else (Inl (STR ''specification has no inputs'')))

lemma *calculate-test-suite-naive-as-io-sequences-with-assumption-check-completeness* :
fixes *M* :: ('a::linorder,'b::linorder,'c) fsm
assumes *observable M'*
and *inputs M' = inputs M*
and *completely-specified M'*
and *size M' ≤ m*
and *calculate-test-suite-naive-as-io-sequences-with-assumption-check M m =*

Inr ts
shows $(L\ M' \subseteq L\ M) \longleftrightarrow$ *pass-io-set-maximal* M' *ts*
⟨*proof*⟩

43.4.2 Greedy Repetition Set Strategy

definition *calculate-test-suite-greedy* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*) *fsm* ⇒ *nat* ⇒ ('*a*,'*b*,'*c*, ('*a* × '*a*) + '*a*) *test-suite* **where**
calculate-test-suite-greedy M m = *calculate-test-suite-for-repetition-sets M m* (*maximal-repetition-sets-from-se*
M)

definition *calculate-test-suite-greedy-as-io-sequences* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*) *fsm* ⇒ *nat* ⇒ ('*b* × '*c*) *list set* **where**
calculate-test-suite-greedy-as-io-sequences M m = *test-suite-to-io-maximal M* (*calculate-test-suite-greedy M m*)

lemma *calculate-test-suite-greedy-completeness* :
fixes *M* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*) *fsm*
assumes *observable M*
and *observable M'*
and *inputs M'* = *inputs M*
and *inputs M* ≠ {}
and *completely-specified M*
and *completely-specified M'*
and *size M'* ≤ *m*
shows $(L\ M' \subseteq L\ M) \longleftrightarrow$ *passes-test-suite M* (*calculate-test-suite-greedy M m*) *M'*
and $(L\ M' \subseteq L\ M) \longleftrightarrow$ *pass-io-set-maximal M'* (*calculate-test-suite-greedy-as-io-sequences M m*)
⟨*proof*⟩

definition *calculate-test-suite-greedy-as-io-sequences-with-assumption-check* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*) *fsm* ⇒ *nat* ⇒ *String.literal* + (('*b* × '*c*) *list set*) **where**
calculate-test-suite-greedy-as-io-sequences-with-assumption-check M m =
(*if inputs M* ≠ {}
then if observable M
then if completely-specified M
then (*Inr* (*test-suite-to-io-maximal M* (*calculate-test-suite-greedy M m*)))
else (*Inl* (*STR* ''*specification is not completely specified*''))
else (*Inl* (*STR* ''*specification is not observable*''))
else (*Inl* (*STR* ''*specification has no inputs*'')))

lemma *calculate-test-suite-greedy-as-io-sequences-with-assumption-check-completeness*
:
fixes *M* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*) *fsm*
assumes *observable M'*
and *inputs M'* = *inputs M*
and *completely-specified M'*

and *size M′ ≤ m*
and *calculate-test-suite-greedy-as-io-sequences-with-assumption-check M m = Inr ts*
shows (*L M′ ⊆ L M*) ⟷ *pass-io-set-maximal M′ ts*
⟨*proof*⟩

end

44 Refined Test Suite Calculation

This theory refines some of the algorithms defined in *Test-Suite-Calculation* using containers from the Containers framework.

theory *Test-Suite-Calculation-Refined*
imports *Test-Suite-Calculation*
../Util-Refined
Deriving.Compare
Containers.Containers
begin

44.1 New Instances

44.1.1 Order on FSMs

instantiation *fsm* :: (*ord*,*ord*,*ord*) *ord*
begin

fun *less-eq-fsm* :: (′*a*,′*b*,′*c*) *fsm* ⇒ (′*a*,′*b*,′*c*) *fsm* ⇒ *bool* **where**
less-eq-fsm M1 M2 =
(*if initial M1* < *initial M2*
then True
else ((*initial M1* = *initial M2*) ∧ (*if set-less-aux* (*states M1*) (*states M2*)
then True
else ((*states M1* = *states M2*) ∧ (*if set-less-aux* (*inputs M1*) (*inputs M2*)
then True
else ((*inputs M1* = *inputs M2*) ∧ (*if set-less-aux* (*outputs M1*) (*outputs M2*)
then True
else ((*outputs M1* = *outputs M2*) ∧ (*set-less-aux* (*transitions M1*) (*transitions M2*) ∨ (*transitions M1*) = (*transitions M2*))))))))))

fun *less-fsm* :: (′*a*,′*b*,′*c*) *fsm* ⇒ (′*a*,′*b*,′*c*) *fsm* ⇒ *bool* **where**
less-fsm a b = (*a* ≤ *b* ∧ *a* ≠ *b*)

instance ⟨*proof*⟩
end

instantiation *fsm* :: (*linorder*,*linorder*,*linorder*) *linorder*

begin

lemma *less-le-not-le-FSM* :
  fixes x :: ($'a$,$'b$,$'c$) *fsm*
  and y :: ($'a$,$'b$,$'c$) *fsm*
shows $(x < y) = (x \leq y \wedge \neg\ y \leq x)$
⟨*proof*⟩

lemma *order-refl-FSM* :
  fixes x :: ($'a$,$'b$,$'c$) *fsm*
  shows $x \leq x$
  ⟨*proof*⟩

lemma *order-trans-FSM* :
  fixes x :: ($'a$,$'b$,$'c$) *fsm*
  fixes y :: ($'a$,$'b$,$'c$) *fsm*
  fixes z :: ($'a$,$'b$,$'c$) *fsm*
  shows $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$
  ⟨*proof*⟩

lemma *antisym-FSM* :
  fixes x :: ($'a$,$'b$,$'c$) *fsm*
  fixes y :: ($'a$,$'b$,$'c$) *fsm*
shows $x \leq y \Longrightarrow y \leq x \Longrightarrow x = y$
  ⟨*proof*⟩

lemma *linear-FSM* :
  fixes x :: ($'a$,$'b$,$'c$) *fsm*
  fixes y :: ($'a$,$'b$,$'c$) *fsm*
shows $x \leq y \vee y \leq x$
  ⟨*proof*⟩

instance
  ⟨*proof*⟩
end

instantiation *fsm* :: (*linorder*,*linorder*,*linorder*) *compare*
begin
fun *compare-fsm* :: ($'a$, $'b$, $'c$) *fsm* ⇒ ($'a$, $'b$, $'c$) *fsm* ⇒ *order* **where**
  compare-fsm x y = *comparator-of x y*

instance
  ⟨*proof*⟩
end

44.1.2 Derived Instances

derive (*eq*) *ceq fsm*

derive (*dlist*) *set-impl fsm*
derive (*assoclist*) *mapping-impl fsm*

derive (*no*) *cenum fsm*
derive (*no*) *ccompare fsm*

44.1.3 Finiteness and Cardinality Instantiations for FSMs

lemma *finiteness-fsm-UNIV* : *finite* (*UNIV* :: ('*a*,'*b*,'*c*) *fsm set*) =
(*finite* (*UNIV* :: '*a set*) ∧ *finite* (*UNIV* :: '*b set*) ∧ *finite* (*UNIV* :: '*c set*))
⟨*proof*⟩

instantiation *fsm* :: (*finite-UNIV*,*finite-UNIV*,*finite-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(('*a*,'*b*,'*c*) *fsm*) (*of-phantom* (*finite-UNIV* :: '*a finite-UNIV*) ∧
of-phantom (*finite-UNIV* :: '*b finite-UNIV*)
∧
of-phantom (*finite-UNIV* :: '*c finite-UNIV*))

instance ⟨*proof*⟩
end

instantiation *fsm* :: (*card-UNIV*,*card-UNIV*,*card-UNIV*) *card-UNIV* **begin**

definition *card-UNIV* = *Phantom*(('*a*,'*b*,'*c*) *fsm*)
(*if CARD*('*a*) = *0* ∨ *CARD*('*b*) = *0* ∨ *CARD*('*c*) = *0*
then 0
else card ((λ(*q*::'*a*, *Q*, *X*::'*b set*, *Y*::'*c set*, *T*). *FSM.create-fsm-from-sets q Q X Y T*) ‘ *UNIV*))
instance ⟨*proof*⟩
end

instantiation *fsm* :: (*type*,*type*,*type*) *cproper-interval* **begin**
definition *cproper-interval-fsm* :: (('*a*,'*b*,'*c*) *fsm*) *proper-interval* **where**
cproper-interval-fsm m1 m2 = *undefined*
instance ⟨*proof*⟩
end

44.2 Updated Code Equations

44.2.1 New Code Equations for *remove-proper-prefixes*

declare [[*code drop*: *remove-proper-prefixes*]]

lemma *remove-proper-prefixes-refined*[*code*] :
 fixes *t* :: (*'a* :: *ccompare*) *list set-rbt*
shows *remove-proper-prefixes* (*RBT-set t*) = (*case ID CCOMPARE*((*'a list*)) *of*
 Some - ⇒ (*if* (*is-empty t*) *then* {} *else set* (*paths* (*from-list* (*RBT-Set2.keys t*))))
|
 None ⇒ *Code.abort* (*STR ''remove-proper-prefixes RBT-set*: *ccompare* = *None''*)
(λ-. *remove-proper-prefixes* (*RBT-set t*)))
 (**is** *?v1* = *?v2*)
⟨*proof*⟩

44.2.2 Special Handling for *set-as-map* on *image*

Avoid creating an intermediate set for (*image f xs*) when evaluating (*set-as-map* (*image f xs*)).

definition *set-as-map-image* :: (*'a1* × *'a2*) *set* ⇒ ((*'a1* × *'a2*) ⇒ (*'b1* × *'b2*)) ⇒ (*'b1* ⇒ *'b2 set option*) **where**
 set-as-map-image xs f = (*set-as-map* (*image f xs*))

definition *dual-set-as-map-image* :: (*'a1* × *'a2*) *set* ⇒ ((*'a1* × *'a2*) ⇒ (*'b1* × *'b2*)) ⇒ ((*'a1* × *'a2*) ⇒ (*'c1* × *'c2*)) ⇒ ((*'b1* ⇒ *'b2 set option*) × (*'c1* ⇒ *'c2 set option*)) **where**
 dual-set-as-map-image xs f1 f2 = (*set-as-map* (*image f1 xs*), *set-as-map* (*image f2 xs*))

lemma *set-as-map-image-code*[*code*] :
 fixes *t* :: (*'a1* ::*ccompare* × *'a2* :: *ccompare*) *set-rbt*
 and *f1* :: (*'a1* × *'a2*) ⇒ (*'b1* :: *ccompare* × *'b2* ::*ccompare*)
shows *set-as-map-image* (*RBT-set t*) *f1* = (*case ID CCOMPARE*((*'a1* × *'a2*)) *of*
 Some - ⇒ *Mapping.lookup*
 (*RBT-Set2.fold* (λ *kv m1* .
 (*case f1 kv of* (*x*,*z*) ⇒ (*case Mapping.lookup m1* (*x*) *of None* ⇒ *Mapping.update* (*x*) {*z*} *m1* | *Some zs* ⇒ *Mapping.update* (*x*) (*Set.insert z zs*) *m1*)))
 t
 Mapping.empty) |
 None ⇒ *Code.abort* (*STR ''set-as-map-image RBT-set*: *ccompare* = *None''*)
 (λ-. *set-as-map-image* (*RBT-set t*) *f1*))
⟨*proof*⟩

lemma *dual-set-as-map-image-code*[*code*] :
fixes *t* :: (*'a1* ::*ccompare* × *'a2* :: *ccompare*) *set-rbt*
and *f1* :: (*'a1* × *'a2*) ⇒ (*'b1* :: *ccompare* × *'b2* ::*ccompare*)
and *f2* :: (*'a1* × *'a2*) ⇒ (*'c1* :: *ccompare* × *'c2* ::*ccompare*)
shows *dual-set-as-map-image* (*RBT-set t*) *f1 f2* = (*case ID CCOMPARE*((*'a1* × *'a2*)) *of*
Some - ⇒ *let mm* = (*RBT-Set2.fold* (λ *kv* (*m1*,*m2*) .
(*case f1 kv of* (*x*,*z*) ⇒ (*case Mapping.lookup m1* (*x*) *of None* ⇒ *Mapping.update* (*x*) {*z*} *m1* | *Some zs* ⇒ *Mapping.update* (*x*) (*Set.insert z zs*) *m1*)
, *case f2 kv of* (*x*,*z*) ⇒ (*case Mapping.lookup m2* (*x*) *of None* ⇒ *Mapping.update* (*x*) {*z*} *m2* | *Some zs* ⇒ *Mapping.update* (*x*) (*Set.insert z zs*) *m2*)))
t
(*Mapping.empty*,*Mapping.empty*))
in (*Mapping.lookup* (*fst mm*), *Mapping.lookup* (*snd mm*)) |
None ⇒ *Code.abort* (*STR ''dual-set-as-map-image RBT-set*: *ccompare* = *None''*)
(λ-. (*dual-set-as-map-image* (*RBT-set t*) *f1 f2*)))
⟨*proof*⟩

44.2.3 New Code Equations for *h*

declare [[*code drop*: *h*]]
lemma *h-refined*[*code*] : *h M* (*q*,*x*)
= (*let m* = *set-as-map-image* (*transitions M*) (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*,*q'*))
in (*case m* (*q*,*x*) *of Some yqs* ⇒ *yqs* | *None* ⇒ {}))
⟨*proof*⟩

44.2.4 New Code Equations for *canonical-separator'*

lemma *canonical-separator'-refined*[*code*] :
fixes *M* :: (*'a*,*'b*,*'c*) *fsm-impl*
shows
FSM-Impl.canonical-separator' M P q1 q2 = (*if FSM-Impl.fsm-impl.initial P* = (*q1*,*q2*)
then
(*let f'* = *set-as-map-image* (*FSM-Impl.fsm-impl.transitions M*) (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*));
f = (λ*qx* . (*case f' qx of Some yqs* ⇒ *yqs* | *None* ⇒ {}));
shifted-transitions' = *shifted-transitions* (*FSM-Impl.fsm-impl.transitions P*);
distinguishing-transitions-lr = *distinguishing-transitions f q1 q2* (*FSM-Impl.fsm-impl.states P*) (*FSM-Impl.fsm-impl.inputs P*);
ts = *shifted-transitions'* ∪ *distinguishing-transitions-lr*
in FSMI
(*Inl* (*q1*,*q2*))
((*image Inl* (*FSM-Impl.fsm-impl.states P*)) ∪ {*Inr q1*, *Inr q2*})
(*FSM-Impl.fsm-impl.inputs M* ∪ *FSM-Impl.fsm-impl.inputs P*)
(*FSM-Impl.fsm-impl.outputs M* ∪ *FSM-Impl.fsm-impl.outputs P*)

(*ts*))
else FSMI
(*Inl* (*q1*,*q2*)) {*Inl* (*q1*,*q2*)} {} {} {})
⟨*proof*⟩

44.2.5 New Code Equations for *calculate-test-paths*

lemma *calculate-test-paths-refined*[*code*] :
calculate-test-paths M m d-reachable-states r-distinguishable-pairs repetition-sets =
(*let*
paths-with-witnesses
= (*image* (λ *q* . (*q*,*m-traversal-paths-with-witness M q repetition-sets m*)) *d-reachable-states*);
get-paths
= *m2f* (*set-as-map paths-with-witnesses*);
PrefixPairTests
= ⋃ *q* ∈ *d-reachable-states* . ⋃ *mrsps* ∈ *get-paths q* . *prefix-pair-tests q mrsps*;
PreamblePrefixTests
= ⋃ *q* ∈ *d-reachable-states* . ⋃ *mrsps* ∈ *get-paths q* . *preamble-prefix-tests q mrsps d-reachable-states*;
PreamblePairTests
= *preamble-pair-tests* (⋃ (*q*,*pw*) ∈ *paths-with-witnesses* . ((λ (*p*,(*rd*,*dr*)) . *dr*) ‘ *pw*)) *r-distinguishable-pairs*;
tests
= *PrefixPairTests* ∪ *PreamblePrefixTests* ∪ *PreamblePairTests*;
tps′
= *m2f-by* ⋃ (*set-as-map-image paths-with-witnesses* (λ (*q*,*p*) . (*q*, *image fst p*)));
dual-maps
= *dual-set-as-map-image tests* (λ (*q*,*p*,*q′*) . (*q*,*p*)) (λ (*q*,*p*,*q′*) . ((*q*,*p*),*q′*));
tps″
= *m2f* (*fst dual-maps*);
tps
= (λ *q* . *tps′ q* ∪ *tps″ q*);
rd-targets
= *m2f* (*snd dual-maps*)
in (*tps*, *rd-targets*))

⟨*proof*⟩

44.2.6 New Code Equations for *prefix-pair-tests*

fun *target′* :: *′state* ⇒ (*′state*, *′input*, *′output*) *path* ⇒ *′state* **where**
target′ q [] = *q* |
target′ q p = *t-target* (*last p*)

lemma *target-refined*[*code*] :
target q p = *target′ q p*

⟨*proof*⟩

declare [[*code drop*: *prefix-pair-tests*]]
lemma *prefix-pair-tests-refined*[*code*] :
fixes *t* :: ((*'a* ::*ccompare*,*'b*::*ccompare*,*'c*::*ccompare*) *traversal-path* × (*'a set* × *'a set*)) *set-rbt*
shows *prefix-pair-tests q* (*RBT-set t*) = (*case ID CCOMPARE*(((*'a*,*'b*,*'c*) *traversal-path* × (*'a set* × *'a set*))) *of*
Some - ⇒ *set*
(*concat* (*map* (λ (*p*,(*rd*,*dr*)) .
(*concat* (*map* (λ (*p1*,*p2*) . [(*q*,*p1*,(*target q p2*)), (*q*,*p2*,(*target q p1*))])
(*filter* (λ (*p1*,*p2*) . (*target q p1*) ≠ (*target q p2*) ∧ (*target q p1*) ∈ *rd* ∧ (*target q p2*) ∈ *rd*) (*prefix-pairs p*)))))
(*RBT-Set2.keys t*))) |
None ⇒ *Code.abort* (*STR ''prefix-pair-tests RBT-set*: *ccompare* = *None''*)
(λ-. (*prefix-pair-tests q* (*RBT-set t*))))
(**is** *prefix-pair-tests q* (*RBT-set t*) = *?C*)
⟨*proof*⟩

44.2.7 New Code Equations for *preamble-prefix-tests*

declare [[*code drop*: *preamble-prefix-tests*]]
lemma *preamble-prefix-tests-refined*[*code*] :
fixes *t1* :: ((*'a* ::*ccompare*,*'b*::*ccompare*,*'c*::*ccompare*) *traversal-path* × (*'a set* × *'a set*)) *set-rbt*
and *t2* :: *'a set-rbt*
shows *preamble-prefix-tests q* (*RBT-set t1*) (*RBT-set t2*) = (*case ID CCOMPARE*(((*'a*,*'b*,*'c*) *traversal-path* × (*'a set* × *'a set*))) *of*
Some - ⇒ (*case ID CCOMPARE*(*'a*) *of*
Some - ⇒ *set* (*concat* (*map* (λ (*p*,(*rd*,*dr*)) .
(*concat* (*map* (λ (*p1*,*q2*) . [(*q*,*p1*,*q2*), (*q2*,[],(*target q p1*))])
(*filter* (λ (*p1*,*q2*) . (*target q p1*) ≠ *q2* ∧ (*target q p1*) ∈ *rd* ∧ *q2* ∈ *rd*)
(*List.product* (*prefixes p*) (*RBT-Set2.keys t2*))))))
(*RBT-Set2.keys t1*))) |
None ⇒ *Code.abort* (*STR ''prefix-pair-tests RBT-set*: *ccompare* = *None''*) (λ-. (*preamble-prefix-tests q* (*RBT-set t1*) (*RBT-set t2*)))) |
None ⇒ *Code.abort* (*STR ''prefix-pair-tests RBT-set*: *ccompare* = *None''*) (λ-. (*preamble-prefix-tests q* (*RBT-set t1*) (*RBT-set t2*))))
(**is** *preamble-prefix-tests q* (*RBT-set t1*) (*RBT-set t2*) = *?C*)
⟨*proof*⟩

end

45 Data Refinement on FSM Representations

This section introduces a refinement of the type of finite state machines for code generation, maintaining mappings to access the transition relation to avoid repeated computations.

theory *FSM-Code-Datatype*
imports *FSM HOL−Library.Mapping Containers.Containers*
begin

45.1 Mappings and Function *h*

fun *list-as-mapping* :: (*'a* × *'c*) *list* ⇒ (*'a*,*'c set*) *mapping* **where**
list-as-mapping xs = (*foldr* (λ (*x*,*z*) *m* . *case Mapping.lookup m x of*
None ⇒ *Mapping.update x* {*z*} *m* |
Some zs ⇒ *Mapping.update x* (*insert z zs*) *m*)
xs
Mapping.empty)

lemma *list-as-mapping-lookup*:
fixes *xs* :: (*'a* × *'c*) *list*
shows (*Mapping.lookup* (*list-as-mapping xs*)) = (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ (*set xs*)) *then Some* {*z* . (*x*,*z*) ∈ (*set xs*)} *else None*)
⟨*proof*⟩

lemma *list-as-mapping-lookup-transitions* :
(*case* (*Mapping.lookup* (*list-as-mapping* (*map* (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*,*q'*)) *ts*)) (*q*,*x*)) *of Some ts* ⇒ *ts* | *None* ⇒ {}) = { (*y*,*q'*) . (*q*,*x*,*y*,*q'*) ∈ *set ts*}
(**is** *?S1* = *?S2*)
⟨*proof*⟩

lemma *list-as-mapping-Nil* :
list-as-mapping [] = *Mapping.empty*
⟨*proof*⟩

definition *set-as-mapping* :: (*'a* × *'c*) *set* ⇒ (*'a*,*'c set*) *mapping* **where**
set-as-mapping s = (*THE m* . *Mapping.lookup m* = (*set-as-map s*))

lemma *set-as-mapping-ob* :
obtains *m* **where** *set-as-mapping s* = *m* **and** *Mapping.lookup m* = *set-as-map s*
⟨*proof*⟩

lemma *set-as-mapping-refined*[*code*] :
fixes *t* :: (*'a* :: *ccompare* × *'c* :: *ccompare*) *set-rbt*
and *xs*:: (*'b* :: *ceq* × *'d* :: *ceq*) *set-dlist*
shows *set-as-mapping* (*RBT-set t*) = (*case ID CCOMPARE*((*'a* × *'c*)) *of*

Some - ⇒ (RBT-Set2.fold (λ (x,z) m . case Mapping.lookup m (x) of
None ⇒ Mapping.update (x) {z} m |
Some zs ⇒ Mapping.update (x) (Set.insert z zs) m)
t
Mapping.empty) |
None ⇒ Code.abort (STR ''set-as-map RBT-set: ccompare = None'')
(λ-. set-as-mapping (RBT-set t)))
(**is** *set-as-mapping (RBT-set t) = ?C1 (RBT-set t)*)
and *set-as-mapping (DList-set xs) = (case ID CEQ(('b × 'd)) of*
Some - ⇒ (DList-Set.fold (λ (x,z) m . case Mapping.lookup m (x) of
None ⇒ Mapping.update (x) {z} m |
Some zs ⇒ Mapping.update (x) (Set.insert z zs) m)
xs
Mapping.empty) |
None ⇒ Code.abort (STR ''set-as-map RBT-set: ccompare = None'')
(λ-. set-as-mapping (DList-set xs)))
(**is** *set-as-mapping (DList-set xs) = ?C2 (DList-set xs)*)
⟨*proof*⟩

fun *h-obs-impl-from-h* :: *(('state × 'input), ('output × 'state) set) mapping ⇒ ('state × 'input, ('output, 'state) mapping) mapping* **where**
h-obs-impl-from-h h' = Mapping.map-values
(λ - yqs . let m' = set-as-mapping yqs;
m'' = Mapping.filter (λ y qs . card qs = 1) m';
m''' = Mapping.map-values (λ - qs . the-elem qs) m''
in m''')
h'

fun *h-obs-impl* :: *(('state × 'input), ('output × 'state) set) mapping ⇒ 'state ⇒ 'input ⇒ 'output ⇒ 'state option* **where**
h-obs-impl h' q x y = (let
tgts = snd ‘ Set.filter (λ(y',q') . y' = y) (case (Mapping.lookup h' (q,x)) of Some ts ⇒ ts | None ⇒ {})
in if card tgts = 1
then Some (the-elem tgts)
else None)

abbreviation(*input*) *h-obs-lookup ≡ (λ h' q x y . (case Mapping.lookup h' (q,x) of Some m ⇒ Mapping.lookup m y | None ⇒ None))*

lemma *h-obs-impl-from-h-invar* : *h-obs-impl h' q x y = h-obs-lookup (h-obs-impl-from-h h') q x y*
(**is** *?A q x y = ?B q x y*)
⟨*proof*⟩

definition *set-as-mapping-image* :: ('*a1* × '*a2*) *set* ⇒ (('*a1* × '*a2*) ⇒ ('*b1* × '*b2*)) ⇒ ('*b1*, '*b2 set*) *mapping* **where**
set-as-mapping-image s f = (*THE m . Mapping.lookup m* = *set-as-map* (*image f s*))

lemma *set-as-mapping-image-ob* :
obtains *m* **where** *set-as-mapping-image s f* = *m* **and** *Mapping.lookup m* = *set-as-map* (*image f s*)
⟨*proof*⟩

lemma *set-as-mapping-image-code*[*code*] :
fixes *t* :: ('*a1* ::*ccompare* × '*a2* :: *ccompare*) *set-rbt*
and *f1* :: ('*a1* × '*a2*) ⇒ ('*b1* :: *ccompare* × '*b2* ::*ccompare*)
and *xs* :: ('*c1* :: *ceq* × '*c2* :: *ceq*) *set-dlist*
and *f2* :: ('*c1* × '*c2*) ⇒ ('*d1* × '*d2*)
shows *set-as-mapping-image* (*RBT-set t*) *f1* = (*case ID CCOMPARE*(('*a1* × '*a2*)) *of*
Some - ⇒ (*RBT-Set2.fold* (λ *kv m1* .
(*case f1 kv of* (*x*,*z*) ⇒ (*case Mapping.lookup m1* (*x*) *of None* ⇒ *Mapping.update* (*x*) {*z*} *m1* | *Some zs* ⇒ *Mapping.update* (*x*) (*Set.insert z zs*) *m1*)))
t
Mapping.empty) |
None ⇒ *Code.abort* (*STR* ''*set-as-map-image RBT-set*: *ccompare* = *None*'')
(λ-. *set-as-mapping-image* (*RBT-set t*) *f1*))
(**is** *set-as-mapping-image* (*RBT-set t*) *f1* = *?C1* (*RBT-set t*))
and *set-as-mapping-image* (*DList-set xs*) *f2* = (*case ID CEQ*(('*c1* × '*c2*)) *of*
Some - ⇒ (*DList-Set.fold* (λ *kv m1* .
(*case f2 kv of* (*x*,*z*) ⇒ (*case Mapping.lookup m1* (*x*) *of None* ⇒ *Mapping.update* (*x*) {*z*} *m1* | *Some zs* ⇒ *Mapping.update* (*x*) (*Set.insert z zs*) *m1*)))
xs
Mapping.empty) |
None ⇒ *Code.abort* (*STR* ''*set-as-map-image DList-set*: *ccompare* = *None*'')
(λ-. *set-as-mapping-image* (*DList-set xs*) *f2*))
(**is** *set-as-mapping-image* (*DList-set xs*) *f2* = *?C2* (*DList-set xs*))
⟨*proof*⟩

45.2 Impl Datatype

The following type extends *fsm-impl* with fields for *h* and *h-obs*.

datatype ('*state*, '*input*, '*output*) *fsm-with-precomputations-impl* =
FSMWPI (*initial-wpi* : '*state*)

(*states-wpi* : *'state set*)
(*inputs-wpi* : *'input set*)
(*outputs-wpi* : *'output set*)
(*transitions-wpi* : (*'state* × *'input* × *'output* × *'state*) *set*)
(*h-wpi* : ((*'state* × *'input*), (*'output* × *'state*) *set*) *mapping*)
(*h-obs-wpi*: (*'state* × *'input*, (*'output*, *'state*) *mapping*) *mapping*)

fun *fsm-with-precomputations-impl-from-list* :: *'a* ⇒ (*'a* × *'b* × *'c* × *'a*) *list* ⇒ (*'a*, *'b*, *'c*) *fsm-with-precomputations-impl* **where**
fsm-with-precomputations-impl-from-list q [] = *FSMWPI q* {*q*} {} {} {} *Mapping.empty Mapping.empty* |
fsm-with-precomputations-impl-from-list q (*t*#*ts*) = (*let ts'* = *set* (*t*#*ts*)
in FSMWPI (*t-source t*)
((*image t-source ts'*) ∪ (*image t-target ts'*))
(*image t-input ts'*)
(*image t-output ts'*)
(*ts'*)
(*list-as-mapping* (*map* (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*,*q'*)) (*t*#*ts*)))
(*h-obs-impl-from-h* (*list-as-mapping* (*map* (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*,*q'*)) (*t*#*ts*)))))

fun *fsm-with-precomputations-impl-from-list'* :: *'a* ⇒ (*'a* × *'b* × *'c* × *'a*) *list* ⇒ (*'a*, *'b*, *'c*) *fsm-with-precomputations-impl* **where**
fsm-with-precomputations-impl-from-list' q [] = *FSMWPI q* {*q*} {} {} {} *Mapping.empty Mapping.empty* |
fsm-with-precomputations-impl-from-list' q (*t*#*ts*) = (*let tsr* = (*remdups* (*t*#*ts*));
h' = (*list-as-mapping* (*map* (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*,*q'*)) *tsr*))
in FSMWPI (*t-source t*)
(*set* (*remdups* ((*map t-source tsr*) @ (*map t-target tsr*))))
(*set* (*remdups* (*map t-input tsr*)))
(*set* (*remdups* (*map t-output tsr*)))
(*set tsr*)
h'
(*h-obs-impl-from-h h'*))

lemma *fsm-impl-from-list-code*[*code*] :
fsm-with-precomputations-impl-from-list q ts = *fsm-with-precomputations-impl-from-list' q ts*
⟨*proof*⟩

45.3 Refined Datatype

Well-formedness now also encompasses the new fields for *h* and *h-obs*.

fun *well-formed-fsm-with-precomputations* :: (*'state*, *'input*, *'output*) *fsm-with-precomputations-impl*

⇒ *bool* **where**
well-formed-fsm-with-precomputations M = (*initial-wpi M* ∈ *states-wpi M*
∧ *finite* (*states-wpi M*)
∧ *finite* (*inputs-wpi M*)
∧ *finite* (*outputs-wpi M*)
∧ *finite* (*transitions-wpi M*)
∧ (∀ *t* ∈ *transitions-wpi M* . *t-source t* ∈ *states-wpi M* ∧
t-input t ∈ *inputs-wpi M* ∧
t-target t ∈ *states-wpi M* ∧
t-output t ∈ *outputs-wpi M*)
∧ (∀ *q x* . (*case* (*Mapping.lookup* (*h-wpi M*) (*q*,*x*)) *of Some ts* ⇒ *ts* | *None* ⇒ {}) = { (*y*,*q'*) . (*q*,*x*,*y*,*q'*) ∈ *transitions-wpi M* })
∧ (∀ *q x y* . *h-obs-impl* (*h-wpi M*) *q x y* = *h-obs-lookup* (*h-obs-wpi M*) *q x y*))

lemma *well-formed-h-set-as-mapping* :
assumes *h-wpi M* = *set-as-mapping-image* (*transitions-wpi M*) (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*,*q'*))
shows (*case* (*Mapping.lookup* (*h-wpi M*) (*q*,*x*)) *of Some ts* ⇒ *ts* | *None* ⇒ {}) = { (*y*,*q'*) . (*q*,*x*,*y*,*q'*) ∈ *transitions-wpi M* }
(**is** *?A q x* = *?B q x*)
⟨*proof*⟩

lemma *well-formed-h-obs-impl-from-h* :
assumes *h-obs-wpi M* = *h-obs-impl-from-h* (*h-wpi M*)
shows *h-obs-impl* (*h-wpi M*) *q x y* = (*h-obs-lookup* (*h-obs-wpi M*) *q x y*)
⟨*proof*⟩

typedef (*'state*, *'input*, *'output*) *fsm-with-precomputations* =
{ *M* :: (*'state*, *'input*, *'output*) *fsm-with-precomputations-impl* . *well-formed-fsm-with-precomputations M*}
morphisms *fsm-with-precomputations-impl-of-fsm-with-precomputations Abs-fsm-with-precomputations*
⟨*proof*⟩

setup-lifting *type-definition-fsm-with-precomputations*

lift-definition *initial-wp* :: (*'state*, *'input*, *'output*) *fsm-with-precomputations* ⇒ *'state* **is** *FSM-Code-Datatype.initial-wpi* ⟨*proof*⟩
lift-definition *states-wp* :: (*'state*, *'input*, *'output*) *fsm-with-precomputations* ⇒ *'state set* **is** *FSM-Code-Datatype.states-wpi* ⟨*proof*⟩
lift-definition *inputs-wp* :: (*'state*, *'input*, *'output*) *fsm-with-precomputations* ⇒ *'input set* **is** *FSM-Code-Datatype.inputs-wpi* ⟨*proof*⟩
lift-definition *outputs-wp* :: (*'state*, *'input*, *'output*) *fsm-with-precomputations* ⇒ *'output set* **is** *FSM-Code-Datatype.outputs-wpi* ⟨*proof*⟩
lift-definition *transitions-wp* ::
(*'state*, *'input*, *'output*) *fsm-with-precomputations* ⇒ (*'state* × *'input* × *'output* × *'state*) *set*
is *FSM-Code-Datatype.transitions-wpi* ⟨*proof*⟩

lift-definition *h-wp* ::
(*'state*, *'input*, *'output*) *fsm-with-precomputations* ⇒ ((*'state* × *'input*), (*'output* × *'state*) *set*) *mapping*
is *FSM-Code-Datatype.h-wpi* ⟨*proof*⟩
lift-definition *h-obs-wp* ::
(*'state*, *'input*, *'output*) *fsm-with-precomputations* ⇒ ((*'state* × *'input*), (*'output*, *'state*) *mapping*) *mapping*
is *FSM-Code-Datatype.h-obs-wpi* ⟨*proof*⟩

lemma *fsm-with-precomputations-initial*: *initial-wp M* ∈ *states-wp M*
⟨*proof*⟩
lemma *fsm-with-precomputations-states-finite*: *finite* (*states-wp M*)
⟨*proof*⟩
lemma *fsm-with-precomputations-inputs-finite*: *finite* (*inputs-wp M*)
⟨*proof*⟩
lemma *fsm-with-precomputations-outputs-finite*: *finite* (*outputs-wp M*)
⟨*proof*⟩
lemma *fsm-with-precomputations-transitions-finite*: *finite* (*transitions-wp M*)
⟨*proof*⟩
lemma *fsm-with-precomputations-transition-props*: *t* ∈ *transitions-wp M* ⟹ *t-source t* ∈ *states-wp M* ∧
t-input t ∈ *inputs-wp M* ∧
t-target t ∈ *states-wp M* ∧
t-output t ∈ *outputs-wp M*
⟨*proof*⟩
lemma *fsm-with-precomputations-h-prop*: (*case* (*Mapping.lookup* (*h-wp M*) (*q*,*x*)) *of Some ts* ⇒ *ts* | *None* ⇒ {}) = { (*y*,*q'*) . (*q*,*x*,*y*,*q'*) ∈ *transitions-wp M* }
⟨*proof*⟩

lemma *fsm-with-precomputations-h-obs-prop*: (*h-obs-lookup* (*h-obs-wp M*) *q x y*) = *h-obs-impl* (*h-wp M*) *q x y*
⟨*proof*⟩

lemma *map-values-empty* : *Mapping.map-values f Mapping.empty* = *Mapping.empty*
⟨*proof*⟩

lift-definition *fsm-with-precomputations-from-list* :: *'a* ⇒ (*'a* × *'b* × *'c* × *'a*) *list* ⇒ (*'a*, *'b*, *'c*) *fsm-with-precomputations*
is *fsm-with-precomputations-impl-from-list*
⟨*proof*⟩

lemma *fsm-with-precomputations-from-list-Nil-simps* :
initial-wp (*fsm-with-precomputations-from-list q* []) = *q*
states-wp (*fsm-with-precomputations-from-list q* []) = {*q*}
inputs-wp (*fsm-with-precomputations-from-list q* []) = {}
outputs-wp (*fsm-with-precomputations-from-list q* []) = {}
transitions-wp (*fsm-with-precomputations-from-list q* []) = {}

⟨*proof*⟩

lemma *fsm-with-precomputations-from-list-Cons-simps* :
initial-wp (*fsm-with-precomputations-from-list q* (*t*#*ts*)) = (*t-source t*)
states-wp (*fsm-with-precomputations-from-list q* (*t*#*ts*)) = ((*image t-source* (*set* (*t*#*ts*))) ∪ (*image t-target* (*set* (*t*#*ts*))))
inputs-wp (*fsm-with-precomputations-from-list q* (*t*#*ts*)) = (*image t-input* (*set* (*t*#*ts*)))
outputs-wp (*fsm-with-precomputations-from-list q* (*t*#*ts*)) = (*image t-output* (*set* (*t*#*ts*)))
transitions-wp (*fsm-with-precomputations-from-list q* (*t*#*ts*)) = (*set* (*t*#*ts*))
⟨*proof*⟩

definition *Fsm-with-precomputations* :: (′*a*,′*b*,′*c*) *fsm-with-precomputations-impl* ⇒ (′*a*,′*b*,′*c*) *fsm-with-precomputations* **where**
Fsm-with-precomputations M = *Abs-fsm-with-precomputations* (*if well-formed-fsm-with-precomputations M then M else FSMWPI undefined* {*undefined*} {} {} {} *Mapping.empty Mapping.empty*)

lemma *fsm-with-precomputations-code-abstype* [*code abstype*] :
Fsm-with-precomputations (*fsm-with-precomputations-impl-of-fsm-with-precomputations M*) = *M*
⟨*proof*⟩

lemma *fsm-with-precomputations-impl-of-fsm-with-precomputations-code* [*code*] :
fsm-with-precomputations-impl-of-fsm-with-precomputations (*fsm-with-precomputations-from-list q ts*) = *fsm-with-precomputations-impl-from-list q ts*
⟨*proof*⟩

definition *FSMWP* :: (′*state*, ′*input*, ′*output*) *fsm-with-precomputations* ⇒ (′*state*, ′*input*, ′*output*) *fsm-impl* **where**
FSMWP M = *FSMI* (*initial-wp M*)
(*states-wp M*)
(*inputs-wp M*)
(*outputs-wp M*)
(*transitions-wp M*)

code-datatype *FSMWP*

45.4 Lifting

declare [[*code drop*: *fsm-impl-from-list*]]
lemma *fsm-impl-from-list*[*code*] :
fsm-impl-from-list q ts = *FSMWP* (*fsm-with-precomputations-from-list q ts*)
⟨*proof*⟩

declare [[*code drop*: *fsm-impl.initial fsm-impl.states fsm-impl.inputs fsm-impl.outputs fsm-impl.transitions*]]
lemma *fsm-impl-FSMWP-initial*[*code*,*simp*] : *fsm-impl.initial* (*FSMWP M*) = *initial-wp M*
⟨*proof*⟩
lemma *fsm-impl-FSMWP-states*[*code*,*simp*] : *fsm-impl.states* (*FSMWP M*) = *states-wp M*
⟨*proof*⟩
lemma *fsm-impl-FSMWP-inputs*[*code*,*simp*] : *fsm-impl.inputs* (*FSMWP M*) = *inputs-wp M*
⟨*proof*⟩
lemma *fsm-impl-FSMWP-outputs*[*code*,*simp*] : *fsm-impl.outputs* (*FSMWP M*) = *outputs-wp M*
⟨*proof*⟩
lemma *fsm-impl-FSMWP-transitions*[*code*,*simp*] : *fsm-impl.transitions* (*FSMWP M*) = *transitions-wp M*
⟨*proof*⟩

lemma *well-formed-FSMWP*: *well-formed-fsm* (*FSMWP M*)
⟨*proof*⟩

declare [[*code drop*: *FSM-Impl.h*]]
lemma *h-with-precomputations-code* [*code*] : *FSM-Impl.h* ((*FSMWP M*)) = (λ (*q*,*x*) . *case Mapping.lookup* (*h-wp M*) (*q*,*x*) *of Some yqs* ⇒ *yqs* | *None* ⇒ {})
⟨*proof*⟩

declare [[*code drop*: *FSM-Impl.h-obs*]]
lemma *h-obs-with-precomputations-code* [*code*] : *FSM-Impl.h-obs* ((*FSMWP M*)) *q x y* = (*h-obs-lookup* (*h-obs-wp M*) *q x y*)
⟨*proof*⟩

fun *filter-states-impl* :: ('*a*,'*b*,'*c*) *fsm-with-precomputations-impl* ⇒ ('*a* ⇒ *bool*) ⇒ ('*a*,'*b*,'*c*) *fsm-with-precomputations-impl* **where**
filter-states-impl M P = (*if P* (*initial-wpi M*)
then (*let*
h′ = *Mapping.filter* (λ (*q*,*x*) *yqs* . *P q*) (*h-wpi M*);
h′′ = *Mapping.map-values* (λ - *yqs* . *Set.filter* (λ (*y*,*q*′) . *P q*′) *yqs*) *h*′
in
FSMWPI (*initial-wpi M*)
(*Set.filter P* (*states-wpi M*))
(*inputs-wpi M*)
(*outputs-wpi M*)

(Set.filter (λ t . P (t-source t) ∧ P (t-target t)) (transitions-wpi M))
h′′
(h-obs-impl-from-h h′′))
else M)

lift-definition *filter-states* :: *('a,'b,'c) fsm-with-precomputations ⇒ ('a ⇒ bool) ⇒ ('a,'b,'c) fsm-with-precomputations*
is *filter-states-impl*
⟨*proof*⟩

lemma *filter-states-simps*:
initial-wp (filter-states M P) = initial-wp M
states-wp (filter-states M P) = (if P (initial-wp M) then Set.filter P (states-wp M) else states-wp M)
inputs-wp (filter-states M P) = inputs-wp M
outputs-wp (filter-states M P) = outputs-wp M
transitions-wp (filter-states M P) = (if P (initial-wp M) then (Set.filter (λ t . P (t-source t) ∧ P (t-target t)) (transitions-wp M)) else transitions-wp M)
⟨*proof*⟩

declare [[*code drop*: *FSM-Impl.filter-states*]]
lemma *filter-states-with-precomputations-code* [*code*] : *FSM-Impl.filter-states ((FSMWP M)) P = FSMWP (filter-states M P)*
⟨*proof*⟩

fun *create-unconnected-fsm-from-fsets-impl* :: *'a ⇒ 'a fset ⇒ 'b fset ⇒ 'c fset ⇒ ('a,'b,'c) fsm-with-precomputations-impl* **where**
create-unconnected-fsm-from-fsets-impl q ns ins outs = FSMWPI q (insert q (fset ns)) (fset ins) (fset outs) {} Mapping.empty Mapping.empty

lift-definition *create-unconnected-fsm-from-fsets* :: *'a ⇒ 'a fset ⇒ 'b fset ⇒ 'c fset ⇒ ('a,'b,'c) fsm-with-precomputations*
is *create-unconnected-fsm-from-fsets-impl*
⟨*proof*⟩

lemma *fsm-with-precomputations-impl-of-code* [*code*] :
fsm-with-precomputations-impl-of-fsm-with-precomputations (create-unconnected-fsm-from-fsets q ns ins outs) = create-unconnected-fsm-from-fsets-impl q ns ins outs
⟨*proof*⟩

lemma *create-unconnected-fsm-from-fsets-simps*:
initial-wp (create-unconnected-fsm-from-fsets q ns ins outs) = q
states-wp (create-unconnected-fsm-from-fsets q ns ins outs) = (insert q (fset ns))

inputs-wp (*create-unconnected-fsm-from-fsets q ns ins outs*) = *fset ins*
outputs-wp (*create-unconnected-fsm-from-fsets q ns ins outs*) = *fset outs*
transitions-wp (*create-unconnected-fsm-from-fsets q ns ins outs*) = {}
⟨*proof*⟩

declare [[*code drop*: *FSM-Impl.create-unconnected-fsm-from-fsets*]]
lemma *create-unconnected-fsm-with-precomputations-code* [*code*] : *FSM-Impl.create-unconnected-fsm-from-fsets q ns ins outs* = *FSMWP* (*create-unconnected-fsm-from-fsets q ns ins outs*)
⟨*proof*⟩

fun *add-transitions-impl* :: ('*a*,'*b*,'*c*) *fsm-with-precomputations-impl* ⇒ ('*a* × '*b* × '*c* × '*a*) *set* ⇒ ('*a*,'*b*,'*c*) *fsm-with-precomputations-impl* **where**
add-transitions-impl M ts = (*if* (∀ *t* ∈ *ts* . *t-source t* ∈ *states-wpi M* ∧ *t-input t* ∈ *inputs-wpi M* ∧ *t-output t* ∈ *outputs-wpi M* ∧ *t-target t* ∈ *states-wpi M*)
then (*let ts'* = ((*transitions-wpi M*) ∪ *ts*);
h' = *set-as-mapping-image ts'* (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*,*q'*))
in FSMWPI
(*initial-wpi M*)
(*states-wpi M*)
(*inputs-wpi M*)
(*outputs-wpi M*)
ts'
h'
(*h-obs-impl-from-h h'*))
else M)

lift-definition *add-transitions* :: ('*a*,'*b*,'*c*) *fsm-with-precomputations* ⇒ ('*a* × '*b* × '*c* × '*a*) *set* ⇒ ('*a*,'*b*,'*c*) *fsm-with-precomputations*
is *add-transitions-impl*
⟨*proof*⟩

lemma *add-transitions-simps*:
initial-wp (*add-transitions M ts*) = *initial-wp M*
states-wp (*add-transitions M ts*) = *states-wp M*
inputs-wp (*add-transitions M ts*) = *inputs-wp M*
outputs-wp (*add-transitions M ts*) = *outputs-wp M*
transitions-wp (*add-transitions M ts*) = (*if* (∀ *t* ∈ *ts* . *t-source t* ∈ *states-wp M* ∧ *t-input t* ∈ *inputs-wp M* ∧ *t-output t* ∈ *outputs-wp M* ∧ *t-target t* ∈ *states-wp M*)
then transitions-wp M ∪ *ts else transitions-wp M*)
⟨*proof*⟩

declare [[*code drop*: *FSM-Impl.add-transitions*]]
lemma *add-transitions-with-precomputations-code* [*code*] : *FSM-Impl.add-transitions*

((*FSMWP M*)) *ts* = *FSMWP* (*add-transitions M ts*)
⟨*proof*⟩

fun *rename-states-impl* :: ('*a*,'*b*,'*c*) *fsm-with-precomputations-impl* ⇒ ('*a* ⇒ '*d*) ⇒ ('*d*,'*b*,'*c*) *fsm-with-precomputations-impl* **where**
rename-states-impl M f = (*let ts* = ((λ*t* . (*f* (*t-source t*), *t-input t*, *t-output t*, *f* (*t-target t*))) ' *transitions-wpi M*);
h' = *set-as-mapping-image ts* (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*,*q'*))
in
FSMWPI (*f* (*initial-wpi M*))
(*f* ' *states-wpi M*)
(*inputs-wpi M*)
(*outputs-wpi M*)
ts
h'
(*h-obs-impl-from-h h'*))

lift-definition *rename-states* :: ('*a*,'*b*,'*c*) *fsm-with-precomputations* ⇒('*a* ⇒ '*d*) ⇒ ('*d*,'*b*,'*c*) *fsm-with-precomputations*
is *rename-states-impl*
⟨*proof*⟩

lemma *rename-states-simps*:
initial-wp (*rename-states M f*) = *f* (*initial-wp M*)
states-wp (*rename-states M f*) = *f* ' *states-wp M*
inputs-wp (*rename-states M f*) = *inputs-wp M*
outputs-wp (*rename-states M f*) = *outputs-wp M*
transitions-wp (*rename-states M f*) = ((λ*t* . (*f* (*t-source t*), *t-input t*, *t-output t*, *f* (*t-target t*))) ' *transitions-wp M*)
⟨*proof*⟩

declare [[*code drop*: *FSM-Impl.rename-states*]]
lemma *rename-states-with-precomputations-code*[*code*] : *FSM-Impl.rename-states* ((*FSMWP M*)) *f* = *FSMWP* (*rename-states M f*)
⟨*proof*⟩

fun *filter-transitions-impl* :: ('*a*,'*b*,'*c*) *fsm-with-precomputations-impl* ⇒ (('*a* × '*b* × '*c* × '*a*) ⇒ *bool*) ⇒ ('*a*,'*b*,'*c*) *fsm-with-precomputations-impl* **where**
filter-transitions-impl M P = (*let ts* = (*Set.filter P* (*transitions-wpi M*));
h' = (*set-as-mapping-image ts* (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*,*q'*)))
in FSMWPI (*initial-wpi M*)
(*states-wpi M*)
(*inputs-wpi M*)
(*outputs-wpi M*)

ts
h′
(*h-obs-impl-from-h h′*))

lift-definition *filter-transitions* :: (*′a,′b,′c*) *fsm-with-precomputations* ⇒ ((*′a* × *′b* × *′c* × *′a*) ⇒ *bool*) ⇒ (*′a,′b,′c*) *fsm-with-precomputations*
is *filter-transitions-impl*
⟨*proof*⟩

lemma *filter-transitions-simps*:
initial-wp (*filter-transitions M P*) = *initial-wp M*
states-wp (*filter-transitions M P*) = *states-wp M*
inputs-wp (*filter-transitions M P*) = *inputs-wp M*
outputs-wp (*filter-transitions M P*) = *outputs-wp M*
transitions-wp (*filter-transitions M P*) = *Set.filter P* (*transitions-wp M*)
⟨*proof*⟩

declare [[*code drop*: *FSM-Impl.filter-transitions*]]
lemma *filter-transitions-with-precomputations-code* [*code*] : *FSM-Impl.filter-transitions* ((*FSMWP M*)) *P* = *FSMWP* (*filter-transitions M P*)
⟨*proof*⟩

fun *initial-singleton-impl* :: (*′a,′b,′c*) *fsm-with-precomputations-impl* ⇒ (*′a,′b,′c*) *fsm-with-precomputations-impl* **where**
initial-singleton-impl M = *FSMWPI* (*initial-wpi M*)
{*initial-wpi M*}
(*inputs-wpi M*)
(*outputs-wpi M*)
{}
Mapping.empty
Mapping.empty

lemma *set-as-mapping-empty* :
set-as-mapping-image {} *f* = *Mapping.empty*
⟨*proof*⟩

lemma *h-obs-from-impl-h* : *h-obs-impl-from-h Mapping.empty* = *Mapping.empty*
⟨*proof*⟩

lift-definition *initial-singleton* :: (*′a,′b,′c*) *fsm-with-precomputations* ⇒ (*′a,′b,′c*) *fsm-with-precomputations*
is *initial-singleton-impl*
⟨*proof*⟩

lemma *initial-singleton-simps*:
initial-wp (*initial-singleton M*) = *initial-wp M*
states-wp (*initial-singleton M*) = {*initial-wp M*}
inputs-wp (*initial-singleton M*) = *inputs-wp M*

outputs-wp (*initial-singleton M*) = *outputs-wp M*
transitions-wp (*initial-singleton M*) = {}
⟨*proof*⟩

declare [[*code drop*: *FSM-Impl.initial-singleton*]]
lemma *initial-singleton-with-precomputations-code*[*code*] : *FSM-Impl.initial-singleton* ((*FSMWP M*)) = *FSMWP* (*initial-singleton M*)
⟨*proof*⟩

fun *canonical-separator'-impl* :: ('a,'b,'c) *fsm-with-precomputations-impl* ⇒ (('a × 'a),'b,'c) *fsm-with-precomputations-impl* ⇒ 'a ⇒ 'a ⇒ (('a × 'a) + 'a,'b,'c) *fsm-with-precomputations-impl* **where**
canonical-separator'-impl M P q1 q2 = (*if initial-wpi P* = (*q1*,*q2*)
then
(*let f'* = *set-as-map* (*image* (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*)) (*transitions-wpi M*));
f = (λ*qx* . (*case f' qx of Some yqs* ⇒ *yqs* | *None* ⇒ {}));
shifted-transitions' = *shifted-transitions* (*transitions-wpi P*);
distinguishing-transitions-lr = *distinguishing-transitions f q1 q2* (*states-wpi P*) (*inputs-wpi P*);
ts = *shifted-transitions'* ∪ *distinguishing-transitions-lr*;
h' = *set-as-mapping-image ts* (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*,*q'*))
in

FSMWPI (*Inl* (*q1*,*q2*))
((*image Inl* (*states-wpi P*)) ∪ {*Inr q1*, *Inr q2*})
(*inputs-wpi M* ∪ *inputs-wpi P*)
(*outputs-wpi M* ∪ *outputs-wpi P*)
ts
h'
(*h-obs-impl-from-h h'*))
else FSMWPI (*Inl* (*q1*,*q2*)) {*Inl* (*q1*,*q2*)} {} {} {} *Mapping.empty Mapping.empty*)

lemma *canonical-separator'-impl-refined*[*code*]:
canonical-separator'-impl M P q1 q2 = (*if initial-wpi P* = (*q1*,*q2*)
then
(*let f'* = *set-as-mapping-image* (*transitions-wpi M*) (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*));
f = (λ*qx* . (*case Mapping.lookup f' qx of Some yqs* ⇒ *yqs* | *None* ⇒ {}));
shifted-transitions' = *shifted-transitions* (*transitions-wpi P*);
distinguishing-transitions-lr = *distinguishing-transitions f q1 q2* (*states-wpi P*) (*inputs-wpi P*);
ts = *shifted-transitions'* ∪ *distinguishing-transitions-lr*;
h' = *set-as-mapping-image ts* (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*,*q'*))
in

FSMWPI (*Inl* (*q1*,*q2*))
((*image Inl* (*states-wpi P*)) ∪ {*Inr q1*, *Inr q2*})
(*inputs-wpi M* ∪ *inputs-wpi P*)
(*outputs-wpi M* ∪ *outputs-wpi P*)

ts
h′
(*h-obs-impl-from-h h′*))
else FSMWPI (*Inl* (*q1*,*q2*)) {*Inl* (*q1*,*q2*)} {} {} {} *Mapping.empty Mapping.empty*)
⟨*proof*⟩

lift-definition *canonical-separator′* :: (′*a*,′*b*,′*c*) *fsm-with-precomputations* ⇒ ((′*a* × ′*a*),′*b*,′*c*) *fsm-with-precomputations* ⇒ ′*a* ⇒ ′*a* ⇒ ((′*a* × ′*a*) + ′*a*,′*b*,′*c*) *fsm-with-precomputations*
is *canonical-separator′-impl*
⟨*proof*⟩

lemma *canonical-separator′-simps* :
initial-wp (*canonical-separator′ M P q1 q2*) = *Inl* (*q1*,*q2*)
states-wp (*canonical-separator′ M P q1 q2*) = (*if initial-wp P* = (*q1*,*q2*) *then* (*image Inl* (*states-wp P*)) ∪ {*Inr q1*, *Inr q2*} *else* {*Inl* (*q1*,*q2*)})
inputs-wp (*canonical-separator′ M P q1 q2*) = (*if initial-wp P* = (*q1*,*q2*) *then inputs-wp M* ∪ *inputs-wp P else* {})
outputs-wp (*canonical-separator′ M P q1 q2*) = (*if initial-wp P* = (*q1*,*q2*) *then outputs-wp M* ∪ *outputs-wp P else* {})
transitions-wp (*canonical-separator′ M P q1 q2*) = (*if initial-wp P* = (*q1*,*q2*) *then shifted-transitions* (*transitions-wp P*) ∪ *distinguishing-transitions* (λ (*q*,*x*) . {*y* . ∃ *q′* . (*q*,*x*,*y*,*q′*) ∈ *transitions-wp M*}) *q1 q2* (*states-wp P*) (*inputs-wp P*) *else* {})
⟨*proof*⟩

declare [[*code drop*: *FSM-Impl.canonical-separator′*]]
lemma *canonical-separator-with-precomputations-code* [*code*] : *FSM-Impl.canonical-separator′* ((*FSMWP M*)) ((*FSMWP P*)) *q1 q2* = *FSMWP* (*canonical-separator′ M P q1 q2*)
⟨*proof*⟩

fun *product-impl* :: (′*a*,′*b*,′*c*) *fsm-with-precomputations-impl* ⇒ (′*d*,′*b*,′*c*) *fsm-with-precomputations-impl* ⇒ (′*a* × ′*d*,′*b*,′*c*) *fsm-with-precomputations-impl* **where**
product-impl A B = (*let ts* = (*image* (λ((*qA*,*x*,*y*,*qA′*), (*qB*,*x′*,*y′*,*qB′*)) . ((*qA*,*qB*),*x*,*y*,(*qA′*,*qB′*))) (*Set.filter* (λ((*qA*,*x*,*y*,*qA′*), (*qB*,*x′*,*y′*,*qB′*)) . *x* = *x′* ∧ *y* = *y′*) (⋃(*image* (λ *tA* . *image* (λ *tB* . (*tA*,*tB*)) (*transitions-wpi B*)) (*transitions-wpi A*)))));
h′ = *set-as-mapping-image ts* (λ(*q*,*x*,*y*,*q′*) . ((*q*,*x*),*y*,*q′*))
in
FSMWPI ((*initial-wpi A*, *initial-wpi B*))
((*states-wpi A*) × (*states-wpi B*))
(*inputs-wpi A* ∪ *inputs-wpi B*)
(*outputs-wpi A* ∪ *outputs-wpi B*)
ts
h′
(*h-obs-impl-from-h h′*))

lift-definition *product* :: (′*a*,′*b*,′*c*) *fsm-with-precomputations* ⇒ (′*d*,′*b*,′*c*) *fsm-with-precomputations*

$\Rightarrow$ ('a × 'd,'b,'c) *fsm-with-precomputations* **is** *product-impl*
⟨*proof*⟩

lemma *product-simps*:
initial-wp (*product A B*) = (*initial-wp A*, *initial-wp B*)
states-wp (*product A B*) = (*states-wp A*) × (*states-wp B*)
inputs-wp (*product A B*) = *inputs-wp A* ∪ *inputs-wp B*
outputs-wp (*product A B*) = *outputs-wp A* ∪ *outputs-wp B*
transitions-wp (*product A B*) = (*image* (λ((*qA*,*x*,*y*,*qA'*), (*qB*,*x'*,*y'*,*qB'*)) . ((*qA*,*qB*),*x*,*y*,(*qA'*,*qB'*))) (*Set.filter* (λ((*qA*,*x*,*y*,*qA'*), (*qB*,*x'*,*y'*,*qB'*)) . *x* = *x'* ∧ *y* = *y'*) (⋃(*image* (λ *tA* . *image* (λ *tB* . (*tA*,*tB*)) (*transitions-wp B*)) (*transitions-wp A*)))))
⟨*proof*⟩

declare [[*code drop*: *FSM-Impl.product*]]
lemma *product-with-precomputations-code* [*code*] : *FSM-Impl.product* ((*FSMWP A*)) ((*FSMWP B*)) = *FSMWP* (*product A B*)
⟨*proof*⟩

fun *from-FSMI-impl* :: ('a,'b,'c) *fsm-with-precomputations-impl* ⇒ 'a ⇒ ('a,'b,'c) *fsm-with-precomputations-impl* **where**
from-FSMI-impl M q = (*if q* ∈ *states-wpi M then FSMWPI q* (*states-wpi M*) (*inputs-wpi M*) (*outputs-wpi M*) (*transitions-wpi M*) (*h-wpi M*) (*h-obs-wpi M*) *else M*)

lift-definition *from-FSMI* :: ('a,'b,'c) *fsm-with-precomputations* ⇒ 'a ⇒ ('a,'b,'c) *fsm-with-precomputations* **is** *from-FSMI-impl*
⟨*proof*⟩

lemma *from-FSMI-simps*:
initial-wp (*from-FSMI M q*) = (*if q* ∈ *states-wp M then q else initial-wp M*)
states-wp (*from-FSMI M q*) = *states-wp M*
inputs-wp (*from-FSMI M q*) = *inputs-wp M*
outputs-wp (*from-FSMI M q*) = *outputs-wp M*
transitions-wp (*from-FSMI M q*) = *transitions-wp M*
⟨*proof*⟩

declare [[*code drop*: *FSM-Impl.from-FSMI*]]
lemma *from-FSMI-with-precomputations-code* [*code*] : *FSM-Impl.from-FSMI* ((*FSMWP M*)) *q* = *FSMWP* (*from-FSMI M q*)
⟨*proof*⟩

end

46 Code Export

This theory exports various functions developed in this library.

theory *Test-Suite-Generator-Code-Export*

imports *EquivalenceTesting/H-Method-Implementations*
EquivalenceTesting/HSI-Method-Implementations
EquivalenceTesting/W-Method-Implementations
EquivalenceTesting/Wp-Method-Implementations
EquivalenceTesting/SPY-Method-Implementations
EquivalenceTesting/SPYH-Method-Implementations
EquivalenceTesting/Partial-S-Method-Implementations
AdaptiveStateCounting/Test-Suite-Calculation-Refined
Prime-Transformation
Prefix-Tree-Refined
EquivalenceTesting/Test-Suite-Representations-Refined
HOL−Library.List-Lexorder
HOL−Library.Code-Target-Nat
HOL−Library.Code-Target-Int
Native-Word.Uint64
FSM-Code-Datatype
begin

46.1 Reduction Testing

definition *generate-reduction-test-suite-naive* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *String.literal* + (*uint64*×*uint64*) *list list* **where**
generate-reduction-test-suite-naive M m = (*case* (*calculate-test-suite-naive-as-io-sequences-with-assumption-c*
M (*nat-of-integer m*)) *of*
Inl err ⇒ *Inl err* |
Inr ts ⇒ *Inr* (*sorted-list-of-set ts*))

definition *generate-reduction-test-suite-greedy* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *String.literal* + (*uint64*×*uint64*) *list list* **where**
generate-reduction-test-suite-greedy M m = (*case* (*calculate-test-suite-greedy-as-io-sequences-with-assumption-*
M (*nat-of-integer m*)) *of*
Inl err ⇒ *Inl err* |
Inr ts ⇒ *Inr* (*sorted-list-of-set ts*))

46.1.1 Fault Detection Capabilities of the Test Harness

The test harness for reduction testing (see https://bitbucket.org/Robert-Sachtleben/an-approach-for-the-verification-and-synthesis-of-complete) applies a test suite to a system under test (SUT) by repeatedly applying each IO-sequence (test case) in the test suite input by input to the SUT until either the test case has been fully applied or the first output is observed that does not correspond to the outputs in the IO-sequence and then checks whether the observed IO-sequence (consisting of a prefix of the test case possibly followed by an IO-pair consisting of the next input in the test case and an output that is not the next output in the test case) is prefix of some test case in the test suite. If such a prefix exists, then the application passes, else it fails and the overall application is aborted, reporting a failure.

The following lemma shows that the SUT (whose behaviour corresponds to an FSM M') conforms to the specification (here FSM M) if and only if the above application procedure does not fail. As the following lemma uses quantification over all possible responses of the SUT to each test case, a further testability hypothesis is required to transfer this result to the actual test application process, which by necessity can only perform a finite number of applications: we assume that some value k exists such that by applying each test case k times, all responses of the SUT to it can be observed.

lemma *reduction-test-harness-soundness* :
fixes *M* :: (*uint64*,*uint64*,*uint64*) *fsm*
assumes *observable M'*
and *FSM.inputs M' = FSM.inputs M*
and *completely-specified M'*
and *size M'* ≤ *nat-of-integer m*
and *generate-reduction-test-suite-greedy M m = Inr ts*
shows (*L M'* ⊆ *L M*) ⟷ (*list-all* (λ *io* . ¬ (∃ *ioPre x y y' ioSuf* . *io* = *ioPre*@[(*x*,*y*)]@*ioSuf* ∧ *ioPre*@[(*x*,*y'*)] ∈ *L M'* ∧ ¬(∃ *ioSuf'* . *ioPre*@[(*x*,*y'*)]@*ioSuf'* ∈ *list.set ts*))) *ts*)
⟨*proof*⟩

46.2 Equivalence Testing

46.2.1 Test Strategy Application and Transformation

fun *apply-method-to-prime* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ ((*uint64*,*uint64*,*uint64*) *fsm* ⇒ *nat* ⇒ (*uint64*×*uint64*) *prefix-tree*) ⇒ (*uint64*×*uint64*) *prefix-tree* **where**
apply-method-to-prime M additionalStates isAlreadyPrime f = (*let*
M' = (*if isAlreadyPrime then M else to-prime-uint64 M*);
m = *size-r M'* + (*nat-of-integer additionalStates*)
in f M' m)

lemma *apply-method-to-prime-completeness* :
fixes *M2* :: ('*a*,*uint64*,*uint64*) *fsm*
assumes ⋀ *M1 m* (*M2* :: ('*a*,*uint64*,*uint64*) *fsm*) .
observable M1 ⟹
observable M2 ⟹
minimal M1 ⟹
minimal M2 ⟹
size-r M1 ≤ *m* ⟹
size M2 ≤ *m* ⟹
FSM.inputs M2 = FSM.inputs M1 ⟹
FSM.outputs M2 = FSM.outputs M1 ⟹
(*L M1 = L M2*) ⟷ ((*L M1* ∩ *set* (*f M1 m*)) = (*L M2* ∩ *set* (*f M1 m*)))
and *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2 = FSM.inputs M1*

and *FSM.outputs M2 = FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1 = states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*apply-method-to-prime M1 additionalStates isAlreadyPrime f*)) = (*L M2* ∩ *set* (*apply-method-to-prime M1 additionalStates isAlreadyPrime f*)))
⟨*proof*⟩

fun *apply-to-prime-and-return-io-lists* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ ((*uint64*,*uint64*,*uint64*) *fsm* ⇒ *nat* ⇒ (*uint64*×*uint64*) *prefix-tree*) ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**
apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime f = (*let M′ = (if isAlreadyPrime then M else to-prime-uint64 M) in*
sorted-list-of-maximal-sequences-in-tree (*test-suite-from-io-tree M′* (*FSM.initial M′*) (*apply-method-to-prime M additionalStates isAlreadyPrime f*)))

lemma *apply-to-prime-and-return-io-lists-completeness* :
fixes *M2* :: (′*a*,*uint64*,*uint64*) *fsm*
assumes ⋀ *M1 m* (*M2* :: (′*a*,*uint64*,*uint64*) *fsm*) .
observable M1 ⟹
observable M2 ⟹
minimal M1 ⟹
minimal M2 ⟹
size-r M1 ≤ *m* ⟹
size M2 ≤ *m* ⟹
FSM.inputs M2 = FSM.inputs M1 ⟹
FSM.outputs M2 = FSM.outputs M1 ⟹
((*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*f M1 m*)) = (*L M2* ∩ *set* (*f M1 m*))))
∧ *finite-tree* (*f M1 m*)
and *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2 = FSM.inputs M1*
and *FSM.outputs M2 = FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1 = states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*apply-to-prime-and-return-io-lists M1 additionalStates isAlreadyPrime f*)
⟨*proof*⟩

fun *apply-to-prime-and-return-input-lists* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ ((*uint64*,*uint64*,*uint64*) *fsm* ⇒ *nat* ⇒ (*uint64*×*uint64*) *prefix-tree*) ⇒ *uint64 list list* **where**

apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime f = *test-suite-to-input-sequences* (*apply-method-to-prime M additionalStates isAlreadyPrime f*)

lemma *apply-to-prime-and-return-input-lists-completeness* :
fixes *M2* :: (*'a*,*uint64*,*uint64*) *fsm*
assumes $\bigwedge$ *M1 m* (*M2* :: (*'a*,*uint64*,*uint64*) *fsm*) .
observable M1 ⟹
observable M2 ⟹
minimal M1 ⟹
minimal M2 ⟹
size-r M1 ≤ *m* ⟹
size M2 ≤ *m* ⟹
FSM.inputs M2 = *FSM.inputs M1* ⟹
FSM.outputs M2 = *FSM.outputs M1* ⟹
((*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*f M1 m*)) = (*L M2* ∩ *set* (*f M1 m*))))
∧ *finite-tree* (*f M1 m*)
and *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ (∀ *xs*∈*list.set* (*apply-to-prime-and-return-input-lists M1 additionalStates isAlreadyPrime f*). ∀ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst io* = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
⟨*proof*⟩

46.2.2 W-Method

definition *w-method-via-h-framework-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**
w-method-via-h-framework-ts M additionalStates isAlreadyPrime = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime w-method-via-h-framework*

lemma *w-method-via-h-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*w-method-via-h-framework-ts M1 additionalStates isAlreadyPrime*)
⟨*proof*⟩

definition *w-method-via-h-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *uint64 list list* **where**
w-method-via-h-framework-input M additionalStates isAlreadyPrime = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime w-method-via-h-framework*

lemma *w-method-via-h-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ (∀ *xs*∈*list.set* (*w-method-via-h-framework-input M1 additionalStates isAlreadyPrime*). ∀ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst io* = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
⟨*proof*⟩

definition *w-method-via-h-framework-2-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**
w-method-via-h-framework-2-ts M additionalStates isAlreadyPrime = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime w-method-via-h-framework-2*

lemma *w-method-via-h-framework-2-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*w-method-via-h-framework-2-ts M1 additionalStates isAlreadyPrime*)
⟨*proof*⟩

definition *w-method-via-h-framework-2-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *uint64 list list* **where**
w-method-via-h-framework-2-input M additionalStates isAlreadyPrime = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime w-method-via-h-framework-2*

lemma *w-method-via-h-framework-2-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*

and *isAlreadyPrime* $\Longrightarrow$ *observable M1* $\wedge$ *minimal M1* $\wedge$ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) $\longleftrightarrow$ ($\forall$ *xs*$\in$*list.set* (*w-method-via-h-framework-2-input M1 additionalStates isAlreadyPrime*). $\forall$ *xs'*$\in$*list.set* (*prefixes xs*). {*io* $\in$ *L M1*. *map fst io* = *xs'*} = {*io* $\in$ *L M2*. *map fst io* = *xs'*})
⟨*proof*⟩

definition *w-method-via-spy-framework-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**
w-method-via-spy-framework-ts M additionalStates isAlreadyPrime = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime w-method-via-spy-framework*

lemma *w-method-via-spy-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* $\Longrightarrow$ *observable M1* $\wedge$ *minimal M1* $\wedge$ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) $\longleftrightarrow$ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*w-method-via-spy-framework-ts M1 additionalStates isAlreadyPrime*)
⟨*proof*⟩

definition *w-method-via-spy-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *uint64 list list* **where**
w-method-via-spy-framework-input M additionalStates isAlreadyPrime = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime w-method-via-spy-framework*

lemma *w-method-via-spy-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* $\Longrightarrow$ *observable M1* $\wedge$ *minimal M1* $\wedge$ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) $\longleftrightarrow$ ($\forall$ *xs*$\in$*list.set* (*w-method-via-spy-framework-input M1 additionalStates isAlreadyPrime*). $\forall$ *xs'*$\in$*list.set* (*prefixes xs*). {*io* $\in$ *L M1*. *map fst io* = *xs'*} = {*io* $\in$ *L M2*. *map fst io* = *xs'*})
⟨*proof*⟩

definition *w-method-via-pair-framework-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**
w-method-via-pair-framework-ts M additionalStates isAlreadyPrime = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime w-method-via-pair-framework*

lemma *w-method-via-pair-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* $\leq$ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* $\Longrightarrow$ *observable M1* $\wedge$ *minimal M1* $\wedge$ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) $\longleftrightarrow$ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*w-method-via-pair-framework-ts M1 additionalStates isAlreadyPrime*)
⟨*proof*⟩

definition *w-method-via-pair-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *uint64 list list* **where**
w-method-via-pair-framework-input M additionalStates isAlreadyPrime = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime w-method-via-pair-framework*

lemma *w-method-via-pair-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* $\leq$ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* $\Longrightarrow$ *observable M1* $\wedge$ *minimal M1* $\wedge$ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) $\longleftrightarrow$ ($\forall$ *xs*∈*list.set* (*w-method-via-pair-framework-input M1 additionalStates isAlreadyPrime*). $\forall$ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst io* = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
⟨*proof*⟩

46.2.3 Wp-Method

definition *wp-method-via-h-framework-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**
wp-method-via-h-framework-ts M additionalStates isAlreadyPrime = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime wp-method-via-h-framework*

lemma *wp-method-via-h-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* $\leq$ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* $\Longrightarrow$ *observable M1* $\wedge$ *minimal M1* $\wedge$ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*

shows $(L\ M1 = L\ M2) \longleftrightarrow$ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*wp-method-via-h-framework-ts M1 additionalStates isAlreadyPrime*)
⟨*proof*⟩

definition *wp-method-via-h-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *uint64 list list* **where**
wp-method-via-h-framework-input M additionalStates isAlreadyPrime = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime wp-method-via-h-framework*

lemma *wp-method-via-h-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows $(L\ M1 = L\ M2) \longleftrightarrow$ (∀ *xs*∈*list.set* (*wp-method-via-h-framework-input M1 additionalStates isAlreadyPrime*). ∀ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst io* = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
⟨*proof*⟩

definition *wp-method-via-spy-framework-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**
wp-method-via-spy-framework-ts M additionalStates isAlreadyPrime = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime wp-method-via-spy-framework*

lemma *wp-method-via-spy-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows $(L\ M1 = L\ M2) \longleftrightarrow$ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*wp-method-via-spy-framework-ts M1 additionalStates isAlreadyPrime*)
⟨*proof*⟩

definition *wp-method-via-spy-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *uint64 list list* **where**
wp-method-via-spy-framework-input M additionalStates isAlreadyPrime = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime wp-method-via-spy-framework*

lemma *wp-method-via-spy-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*

and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ (∀ *xs*∈*list.set* (*wp-method-via-spy-framework-input M1 additionalStates isAlreadyPrime*). ∀ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst io* = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
⟨*proof*⟩

46.2.4 HSI-Method

definition *hsi-method-via-h-framework-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**
hsi-method-via-h-framework-ts M additionalStates isAlreadyPrime = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime hsi-method-via-h-framework*

lemma *hsi-method-via-h-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*hsi-method-via-h-framework-ts M1 additionalStates isAlreadyPrime*)
⟨*proof*⟩

definition *hsi-method-via-h-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *uint64 list list* **where**
hsi-method-via-h-framework-input M additionalStates isAlreadyPrime = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime hsi-method-via-h-framework*

lemma *hsi-method-via-h-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ (∀ *xs*∈*list.set* (*hsi-method-via-h-framework-input M1 additionalStates isAlreadyPrime*). ∀ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst io* = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
⟨*proof*⟩

definition *hsi-method-via-spy-framework-ts* :: (*uint64* ,*uint64* ,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ ((*uint64* × *uint64*)× *bool*) *list list* **where**
hsi-method-via-spy-framework-ts M additionalStates isAlreadyPrime = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime hsi-method-via-spy-framework*

lemma *hsi-method-via-spy-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*hsi-method-via-spy-framework-ts M1 additionalStates isAlreadyPrime*)
⟨*proof*⟩

definition *hsi-method-via-spy-framework-input* :: (*uint64* ,*uint64* ,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *uint64 list list* **where**
hsi-method-via-spy-framework-input M additionalStates isAlreadyPrime = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime hsi-method-via-spy-framework*

lemma *hsi-method-via-spy-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ (∀ *xs*∈*list.set* (*hsi-method-via-spy-framework-input M1 additionalStates isAlreadyPrime*). ∀ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst io* = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
⟨*proof*⟩

definition *hsi-method-via-pair-framework-ts* :: (*uint64* ,*uint64* ,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ ((*uint64* × *uint64*)× *bool*) *list list* **where**
hsi-method-via-pair-framework-ts M additionalStates isAlreadyPrime = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime hsi-method-via-pair-framework*

lemma *hsi-method-via-pair-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*

and *isAlreadyPrime* $\Longrightarrow$ *observable M1* $\wedge$ *minimal M1* $\wedge$ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) $\longleftrightarrow$ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*hsi-method-via-pair-framework-ts M1 additionalStates isAlreadyPrime*)
⟨*proof*⟩

definition *hsi-method-via-pair-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *uint64 list list* **where**
hsi-method-via-pair-framework-input M additionalStates isAlreadyPrime = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime hsi-method-via-pair-framework*

lemma *hsi-method-via-pair-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* $\Longrightarrow$ *observable M1* $\wedge$ *minimal M1* $\wedge$ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) $\longleftrightarrow$ (∀ *xs*∈*list.set* (*hsi-method-via-pair-framework-input M1 additionalStates isAlreadyPrime*). ∀ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst io* = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
⟨*proof*⟩

46.2.5 H-Method

definition *h-method-via-h-framework-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**
h-method-via-h-framework-ts M additionalStates isAlreadyPrime c b = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime* (λ *M m* . *h-method-via-h-framework M m c b*)

lemma *h-method-via-h-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* $\Longrightarrow$ *observable M1* $\wedge$ *minimal M1* $\wedge$ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) $\longleftrightarrow$ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*h-method-via-h-framework-ts M1 additionalStates isAlreadyPrime c b*)
⟨*proof*⟩

definition *h-method-via-h-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ *uint64 list list* **where**
h-method-via-h-framework-input M additionalStates isAlreadyPrime c b = *ap-*

ply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime (λ *M m* . *h-method-via-h-framework M m c b*)

lemma *h-method-via-h-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ (∀ *xs*∈*list.set* (*h-method-via-h-framework-input M1 additionalStates isAlreadyPrime c b*). ∀ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst io* = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
⟨*proof*⟩

definition *h-method-via-pair-framework-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**
h-method-via-pair-framework-ts M additionalStates isAlreadyPrime = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime h-method-via-pair-framework*

lemma *h-method-via-pair-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*h-method-via-pair-framework-ts M1 additionalStates isAlreadyPrime*)
⟨*proof*⟩

definition *h-method-via-pair-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *uint64 list list* **where**
h-method-via-pair-framework-input M additionalStates isAlreadyPrime = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime h-method-via-pair-framework*

lemma *h-method-via-pair-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*

and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ (∀ *xs*∈*list.set* (*h-method-via-pair-framework-input M1 additionalStates isAlreadyPrime*). ∀ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst io* = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
⟨*proof*⟩

definition *h-method-via-pair-framework-2-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *bool* ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**
h-method-via-pair-framework-2-ts M additionalStates isAlreadyPrime c = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime* (λ *M m* . *h-method-via-pair-framework-2 M m c*)

lemma *h-method-via-pair-framework-2-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*h-method-via-pair-framework-2-ts M1 additionalStates isAlreadyPrime c*)
⟨*proof*⟩

definition *h-method-via-pair-framework-2-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *bool* ⇒ *uint64 list list* **where**
h-method-via-pair-framework-2-input M additionalStates isAlreadyPrime c = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime* (λ *M m* . *h-method-via-pair-framework-2 M m c*)

lemma *h-method-via-pair-framework-2-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ (∀ *xs*∈*list.set* (*h-method-via-pair-framework-2-input M1 additionalStates isAlreadyPrime c*). ∀ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst io* = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
⟨*proof*⟩

definition *h-method-via-pair-framework-3-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**

h-method-via-pair-framework-3-ts M additionalStates isAlreadyPrime c1 c2 = apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime (λ *M m . h-method-via-pair-framework-3 M m c1 c2*)

lemma *h-method-via-pair-framework-3-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* $\leq$ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2 = FSM.inputs M1*
and *FSM.outputs M2 = FSM.outputs M1*
and *isAlreadyPrime* $\Longrightarrow$ *observable M1* $\wedge$ *minimal M1* $\wedge$ *reachable-states M1 = states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1 = L M2*) $\longleftrightarrow$ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*h-method-via-pair-framework-3-ts M1 additionalStates isAlreadyPrime c1 c2*)
⟨*proof*⟩

definition *h-method-via-pair-framework-3-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ *uint64 list list* **where**
h-method-via-pair-framework-3-input M additionalStates isAlreadyPrime c1 c2 = apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime (λ *M m . h-method-via-pair-framework-3 M m c1 c2*)

lemma *h-method-via-pair-framework-3-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* $\leq$ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2 = FSM.inputs M1*
and *FSM.outputs M2 = FSM.outputs M1*
and *isAlreadyPrime* $\Longrightarrow$ *observable M1* $\wedge$ *minimal M1* $\wedge$ *reachable-states M1 = states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1 = L M2*) $\longleftrightarrow$ ($\forall$ *xs*$\in$*list.set* (*h-method-via-pair-framework-3-input M1 additionalStates isAlreadyPrime c1 c2*). $\forall$ *xs'*$\in$*list.set* (*prefixes xs*). {*io* $\in$ *L M1*. *map fst io = xs'*} = {*io* $\in$ *L M2*. *map fst io = xs'*})
⟨*proof*⟩

46.2.6 SPY-Method

definition *spy-method-via-h-framework-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**
spy-method-via-h-framework-ts M additionalStates isAlreadyPrime = apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime spy-method-via-h-framework

lemma *spy-method-via-h-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* $\leq$ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2 = FSM.inputs M1*

and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*spy-method-via-h-framework-ts M1 additionalStates isAlreadyPrime*)
⟨*proof*⟩

definition *spy-method-via-h-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *uint64 list list* **where**
spy-method-via-h-framework-input M additionalStates isAlreadyPrime = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime spy-method-via-h-framework*

lemma *spy-method-via-h-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ (∀ *xs*∈*list.set* (*spy-method-via-h-framework-input M1 additionalStates isAlreadyPrime*). ∀ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst io* = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
⟨*proof*⟩

definition *spy-method-via-spy-framework-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**
spy-method-via-spy-framework-ts M additionalStates isAlreadyPrime = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime spy-method-via-spy-framework*

lemma *spy-method-via-spy-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*spy-method-via-spy-framework-ts M1 additionalStates isAlreadyPrime*)
⟨*proof*⟩

definition *spy-method-via-spy-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *uint64 list list* **where**
spy-method-via-spy-framework-input M additionalStates isAlreadyPrime = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime spy-method-via-spy-framework*

lemma *spy-method-via-spy-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ (∀ *xs*∈*list.set* (*spy-method-via-spy-framework-input M1 additionalStates isAlreadyPrime*). ∀ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst io* = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
⟨*proof*⟩

46.2.7 SPYH-Method

definition *spyh-method-via-h-framework-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**
spyh-method-via-h-framework-ts M additionalStates isAlreadyPrime c b = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime* (λ *M m* . *spyh-method-via-h-framework M m c b*)

lemma *spyh-method-via-h-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*spyh-method-via-h-framework-ts M1 additionalStates isAlreadyPrime c b*)
⟨*proof*⟩

definition *spyh-method-via-h-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ *uint64 list list* **where**
spyh-method-via-h-framework-input M additionalStates isAlreadyPrime c b = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime* (λ *M m* . *spyh-method-via-h-framework M m c b*)

lemma *spyh-method-via-h-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1*

= *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ (∀ *xs*∈*list.set* (*spyh-method-via-h-framework-input M1 additionalStates isAlreadyPrime c b*). ∀ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst io* = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
⟨*proof*⟩

definition *spyh-method-via-spy-framework-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**
spyh-method-via-spy-framework-ts M additionalStates isAlreadyPrime c b = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime* (λ *M m* . *spyh-method-via-spy-framework M m c b*)

lemma *spyh-method-via-spy-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*spyh-method-via-spy-framework-ts M1 additionalStates isAlreadyPrime c b*)
⟨*proof*⟩

definition *spyh-method-via-spy-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ *uint64 list list* **where**
spyh-method-via-spy-framework-input M additionalStates isAlreadyPrime c b = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime* (λ *M m* . *spyh-method-via-spy-framework M m c b*)

lemma *spyh-method-via-spy-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ (∀ *xs*∈*list.set* (*spyh-method-via-spy-framework-input M1 additionalStates isAlreadyPrime c b*). ∀ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst io* = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
⟨*proof*⟩

46.2.8 Partial S-Method

definition *partial-s-method-via-h-framework-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**
partial-s-method-via-h-framework-ts M additionalStates isAlreadyPrime c b = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime* (λ *M m* . *partial-s-method-via-h-framework M m c b*)

lemma *partial-s-method-via-h-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*partial-s-method-via-h-framework-ts M1 additionalStates isAlreadyPrime c b*)
⟨*proof*⟩

definition *partial-s-method-via-h-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ *uint64 list list* **where**
partial-s-method-via-h-framework-input M additionalStates isAlreadyPrime c b = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime* (λ *M m* . *partial-s-method-via-h-framework M m c b*)

lemma *partial-s-method-via-h-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ (∀ *xs*∈*list.set* (*partial-s-method-via-h-framework-input M1 additionalStates isAlreadyPrime c b*). ∀ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst io* = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
⟨*proof*⟩

46.3 New Instances

lemma *finiteness-fset-UNIV* : *finite* (*UNIV* :: *'a fset set*) = *finite* (*UNIV* :: *'a set*)
⟨*proof*⟩

instantiation *fset* :: (*finite-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*'a fset*) (*of-phantom* (*finite-UNIV* :: *'a finite-UNIV*))
instance ⟨*proof*⟩

end

derive (*eq*) *ceq fset*
derive (*no*) *cenum fset*
derive (*no*) *ccompare fset*
derive (*dlist*) *set-impl fset*

instantiation *fset* :: (*type*) *cproper-interval* **begin**
definition *cproper-interval-fset* :: ((*'a*) *fset*) *proper-interval*
  where *cproper-interval-fset - - = undefined*
instance ⟨*proof*⟩
end

lemma *card-fPow*: *card* (*Pow* (*fset A*)) = *2* ^ *card* (*fset A*)
  ⟨*proof*⟩

lemma *finite-sets-finite-univ* :
  assumes *finite* (*UNIV* :: *'a set*)
  shows *finite* (*xs* :: *'a set*)
  ⟨*proof*⟩

lemma *card-UNIV-fset*: *CARD*(*'a fset*) = (*if CARD*(*'a*) = *0 then 0 else 2* ^ *CARD*(*'a*))
  ⟨*proof*⟩

instantiation *fset* :: (*card-UNIV*) *card-UNIV* **begin**
definition *card-UNIV = Phantom*(*'a fset*)
  (*let c = of-phantom* (*card-UNIV* :: *'a card-UNIV*) *in if c = 0 then 0 else 2* ^ *c*)
instance ⟨*proof*⟩
end

derive (*choose*) *mapping-impl fset*

lemma *uint64-range* : *range nat-of-uint64* = {..<*2* ^ *64*}
⟨*proof*⟩

lemma *card-UNIV-uint64*: *CARD*(*uint64*) = *2*^*64*
⟨*proof*⟩

lemma *nat-of-uint64-bij-betw* : *bij-betw nat-of-uint64* (*UNIV* :: *uint64 set*) {..<*2* ^ *64*}
  ⟨*proof*⟩

lemma *uint64-UNIV* : (*UNIV* :: *uint64 set*) = *uint64-of-nat* ‘ {..<*2* ^ *64*}
  ⟨*proof*⟩

lemma *uint64-of-nat-bij-betw* : *bij-betw uint64-of-nat* {..<*2* ^ *64*} (*UNIV* :: *uint64 set*)
⟨*proof*⟩

lemma *uint64-finite* : *finite* (*UNIV* :: *uint64 set*)
⟨*proof*⟩

instantiation *uint64* :: *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*uint64*) *True*
instance ⟨*proof*⟩
end

instantiation *uint64* :: *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*(*uint64*) (*2*^*64*)
instance
⟨*proof*⟩
end

instantiation *uint64* :: *compare*
begin
definition *compare-uint64* :: *uint64* ⇒ *uint64* ⇒ *order* **where**
compare-uint64 x y = (*case* (*x* < *y*, *x* = *y*) *of* (*True*,-) ⇒ *Lt* | (*False*,*True*) ⇒ *Eq* | (*False*,*False*) ⇒ *Gt*)

instance
⟨*proof*⟩
end

instantiation *uint64* :: *ccompare*
begin
definition *ccompare-uint64* :: (*uint64* ⇒ *uint64* ⇒ *order*) *option* **where**
ccompare-uint64 = *Some compare*

instance ⟨*proof*⟩
end

derive (*eq*) *ceq uint64*
derive (*no*) *cenum uint64*
derive (*rbt*) *set-impl uint64*
derive (*rbt*) *mapping-impl uint64*

instantiation *uint64* :: *proper-interval* **begin**
fun *proper-interval-uint64* :: *uint64 proper-interval*

where
proper-interval-uint64 None None = True |
proper-interval-uint64 None (*Some y*) = (*y* > *0*)|
proper-interval-uint64 (*Some x*) *None* = (*x* ≠ *uint64-of-nat* (*2^64−1*)) |
proper-interval-uint64 (*Some x*) (*Some y*) = (*x* < *y* ∧ *x*+*1* < *y*)

instance ⟨*proof*⟩
end

instantiation *uint64* :: *cproper-interval* **begin**
definition *cproper-interval* = (*proper-interval* :: *uint64 proper-interval*)
instance
⟨*proof*⟩
end

46.4 Exports

fun *fsm-from-list-uint64* :: *uint64* ⇒ (*uint64* × *uint64* × *uint64* × *uint64*) *list* ⇒ (*uint64*, *uint64*, *uint64*) *fsm*
where *fsm-from-list-uint64 q ts* = *fsm-from-list q ts*

fun *fsm-from-list-integer* :: *integer* ⇒ (*integer* × *integer* × *integer* × *integer*) *list* ⇒ (*integer*, *integer*, *integer*) *fsm*
where *fsm-from-list-integer q ts* = *fsm-from-list q ts*

export-code *Inl*
fsm-from-list
fsm-from-list-uint64
fsm-from-list-integer
size
to-prime
make-observable
rename-states
index-states
restrict-to-reachable-states
integer-of-nat
generate-reduction-test-suite-naive
generate-reduction-test-suite-greedy
w-method-via-h-framework-ts
w-method-via-h-framework-input
w-method-via-h-framework-2-ts
w-method-via-h-framework-2-input
w-method-via-spy-framework-ts

w-method-via-spy-framework-input
w-method-via-pair-framework-ts
w-method-via-pair-framework-input
wp-method-via-h-framework-ts
wp-method-via-h-framework-input
wp-method-via-spy-framework-ts
wp-method-via-spy-framework-input
hsi-method-via-h-framework-ts
hsi-method-via-h-framework-input
hsi-method-via-spy-framework-ts
hsi-method-via-spy-framework-input
hsi-method-via-pair-framework-ts
hsi-method-via-pair-framework-input
h-method-via-h-framework-ts
h-method-via-h-framework-input
h-method-via-pair-framework-ts
h-method-via-pair-framework-input
h-method-via-pair-framework-2-ts
h-method-via-pair-framework-2-input
h-method-via-pair-framework-3-ts
h-method-via-pair-framework-3-input
spy-method-via-h-framework-ts
spy-method-via-h-framework-input
spy-method-via-spy-framework-ts
spy-method-via-spy-framework-input
spyh-method-via-h-framework-ts
spyh-method-via-h-framework-input
spyh-method-via-spy-framework-ts
spyh-method-via-spy-framework-input
partial-s-method-via-h-framework-ts
partial-s-method-via-h-framework-input
in *Haskell* **module-name** *GeneratedCode* **file-prefix** *haskell-export*

export-code *Inl*
fsm-from-list
fsm-from-list-uint64
fsm-from-list-integer
size
to-prime
make-observable
rename-states
index-states
restrict-to-reachable-states
integer-of-nat
generate-reduction-test-suite-naive
generate-reduction-test-suite-greedy
w-method-via-h-framework-ts
w-method-via-h-framework-input

w-method-via-h-framework-2-ts
w-method-via-h-framework-2-input
w-method-via-spy-framework-ts
w-method-via-spy-framework-input
w-method-via-pair-framework-ts
w-method-via-pair-framework-input
wp-method-via-h-framework-ts
wp-method-via-h-framework-input
wp-method-via-spy-framework-ts
wp-method-via-spy-framework-input
hsi-method-via-h-framework-ts
hsi-method-via-h-framework-input
hsi-method-via-spy-framework-ts
hsi-method-via-spy-framework-input
hsi-method-via-pair-framework-ts
hsi-method-via-pair-framework-input
h-method-via-h-framework-ts
h-method-via-h-framework-input
h-method-via-pair-framework-ts
h-method-via-pair-framework-input
h-method-via-pair-framework-2-ts
h-method-via-pair-framework-2-input
h-method-via-pair-framework-3-ts
h-method-via-pair-framework-3-input
spy-method-via-h-framework-ts
spy-method-via-h-framework-input
spy-method-via-spy-framework-ts
spy-method-via-spy-framework-input
spyh-method-via-h-framework-ts
spyh-method-via-h-framework-input
spyh-method-via-spy-framework-ts
spyh-method-via-spy-framework-input
partial-s-method-via-h-framework-ts
partial-s-method-via-h-framework-input
in *Scala* **module-name** *GeneratedCode* **file-prefix** *scala-export* (*case-insensitive*)

export-code *Inl*
fsm-from-list
fsm-from-list-uint64
fsm-from-list-integer
size
to-prime
make-observable
rename-states
index-states
restrict-to-reachable-states
integer-of-nat
generate-reduction-test-suite-naive

generate-reduction-test-suite-greedy
w-method-via-h-framework-ts
w-method-via-h-framework-input
w-method-via-h-framework-2-ts
w-method-via-h-framework-2-input
w-method-via-spy-framework-ts
w-method-via-spy-framework-input
w-method-via-pair-framework-ts
w-method-via-pair-framework-input
wp-method-via-h-framework-ts
wp-method-via-h-framework-input
wp-method-via-spy-framework-ts
wp-method-via-spy-framework-input
hsi-method-via-h-framework-ts
hsi-method-via-h-framework-input
hsi-method-via-spy-framework-ts
hsi-method-via-spy-framework-input
hsi-method-via-pair-framework-ts
hsi-method-via-pair-framework-input
h-method-via-h-framework-ts
h-method-via-h-framework-input
h-method-via-pair-framework-ts
h-method-via-pair-framework-input
h-method-via-pair-framework-2-ts
h-method-via-pair-framework-2-input
h-method-via-pair-framework-3-ts
h-method-via-pair-framework-3-input
spy-method-via-h-framework-ts
spy-method-via-h-framework-input
spy-method-via-spy-framework-ts
spy-method-via-spy-framework-input
spyh-method-via-h-framework-ts
spyh-method-via-h-framework-input
spyh-method-via-spy-framework-ts
spyh-method-via-spy-framework-input
partial-s-method-via-h-framework-ts
partial-s-method-via-h-framework-input
in *SML* **module-name** *GeneratedCode* **file-prefix** *sml-export*

export-code *Inl*
fsm-from-list
fsm-from-list-uint64
fsm-from-list-integer
size
to-prime
make-observable
rename-states
index-states

restrict-to-reachable-states
integer-of-nat
generate-reduction-test-suite-naive
generate-reduction-test-suite-greedy
w-method-via-h-framework-ts
w-method-via-h-framework-input
w-method-via-h-framework-2-ts
w-method-via-h-framework-2-input
w-method-via-spy-framework-ts
w-method-via-spy-framework-input
w-method-via-pair-framework-ts
w-method-via-pair-framework-input
wp-method-via-h-framework-ts
wp-method-via-h-framework-input
wp-method-via-spy-framework-ts
wp-method-via-spy-framework-input
hsi-method-via-h-framework-ts
hsi-method-via-h-framework-input
hsi-method-via-spy-framework-ts
hsi-method-via-spy-framework-input
hsi-method-via-pair-framework-ts
hsi-method-via-pair-framework-input
h-method-via-h-framework-ts
h-method-via-h-framework-input
h-method-via-pair-framework-ts
h-method-via-pair-framework-input
h-method-via-pair-framework-2-ts
h-method-via-pair-framework-2-input
h-method-via-pair-framework-3-ts
h-method-via-pair-framework-3-input
spy-method-via-h-framework-ts
spy-method-via-h-framework-input
spy-method-via-spy-framework-ts
spy-method-via-spy-framework-input
spyh-method-via-h-framework-ts
spyh-method-via-h-framework-input
spyh-method-via-spy-framework-ts
spyh-method-via-spy-framework-input
partial-s-method-via-h-framework-ts
partial-s-method-via-h-framework-input
in *OCaml* **module-name** *GeneratedCode* **file-prefix** *ocaml-export*

end

References

[1] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–186, Mar.

1978.

[2] R. Dorofeeva, K. El-Fakih, and N. Yevtushenko. An improved conformance testing method. In F. Wang, editor, *Formal Techniques for Networked and Distributed Systems - FORTE 2005, 25th IFIP WG 6.1 International Conference, Taipei, Taiwan, October 2-5, 2005, Proceedings*, volume 3731 of *Lecture Notes in Computer Science*, pages 204–218. Springer, 2005. ISBN 3-540-29189-X. doi: 10.1007/11562436_16. URL https://doi.org/10.1007/11562436_16.

[3] G. Luo, A. Petrenko, and G. v. Bochmann. Selecting test sequences for partially-specified nondeterministic finite state machines. In T. Mizuno, T. Higashino, and N. Shiratori, editors, *Protocol Test Systems: 7th workshop 7th IFIP WG 6.1 international workshop on protocol text systems*, IFIP - The International Federation for Information Processing, pages 95–110. Springer US. ISBN 978-0-387-34883-4. doi: 10.1007/978-0-387-34883-4_6. URL https://doi.org/10.1007/978-0-387-34883-4_6.

[4] G. Luo, G. Bochmann, and A. Petrenko. Test selection based on communicating nondeterministic finite-state machines using a generalized wp-method. *IEEE Transactions on Software Engineering*, 20(2): 149–162, 1994. ISSN 0098-5589. doi: 10.1109/32.265636.

[5] J. Peleska and W.-l. Huang. *Test Automation - Foundations and Applications of Model-based Testing*. University of Bremen, January 2019. Lecture notes, available under http://www.informatik.uni-bremen.de/agbs/jp/papers/test-automation-huang-peleska.pdf.

[6] A. Petrenko and N. Yevtushenko. Adaptive testing of nondeterministic systems with FSM. In *15th International IEEE Symposium on High-Assurance Systems Engineering, HASE 2014, Miami Beach, FL, USA, January 9-11, 2014*, pages 224–228, 2014. doi: 10.1109/HASE.2014.39. URL http://dx.doi.org/10.1109/HASE.2014.39.

[7] R. Sachtleben. Formalisation of an adaptive state counting algorithm. *Archive of Formal Proofs*, Aug. 2019. ISSN 2150-914x. http://isa-afp.org/entries/Adaptive_State_Counting.html, Formal proof development.

[8] R. Sachtleben. An approach for the verification and synthesis of complete test generation algorithms for finite state machines. 2022. doi: 10.26092/elib/1665.

[9] R. Sachtleben, R. M. Hierons, W.-l. Huang, and J. Peleska. A mechanised proof of an adaptive state counting algorithm. In C. Gaston, N. Kosmatov, and P. Le Gall, editors, *Testing Software and Systems*,

pages 176–193, Cham, 2019. Springer International Publishing. ISBN 978-3-030-31280-0.

[10] A. Simão, A. Petrenko, and N. Yevtushenko. On reducing test length for FSMs with extra states. *Software Testing, Verification and Reliability*, 22(6):435–454, Sept. 2012. ISSN 1099-1689. doi: 10.1002/stvr.452. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.452.

[11] M. Soucha and K. Bogdanov. SPYH-method: An improvement in testing of finite-state machines. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 194–203, 2018. doi: 10.1109/ICSTW.2018.00050.