

Formalisation of an Adaptive State Counting Algorithm

Robert Sachtleben

February 6, 2026

Abstract

This entry provides a formalisation of a refinement of an adaptive state counting algorithm, used to test for reduction between finite state machines. The algorithm has been originally presented by Hierons in [2] and was slightly refined by Sachtleben et al. in [3]. Definitions for finite state machines and adaptive test cases are given and many useful theorems are derived from these. The algorithm is formalised using mutually recursive functions, for which it is proven that the generated test suite is sufficient to test for reduction against finite state machines of a certain fault domain. Additionally, the algorithm is specified in a simple WHILE-language and its correctness is shown using Hoare-logic.

Contents

1	Finite state machines	2
1.1	FSMs as transition systems	2
1.2	Language	2
1.3	Product machine for language intersection	3
1.4	Required properties	4
1.5	States reached by a given IO-sequence	6
1.6	D-reachability	8
1.7	Deterministic state cover	9
1.8	IO reduction	10
1.9	Language subsets for input sequences	10
1.10	Sequences to failures	12
1.11	Minimal sequence to failure extending	12
1.12	Complete test suite derived from the product machine	13
2	Product machines with an additional fail state	13
2.1	Sequences to failure in the product machine	17
3	Adaptive test cases	19
3.1	Properties of ATC-reactions	20
3.2	Applicability	20
3.3	Application function IO	21
3.4	R-distinguishability	21
3.5	Response sets	22
3.6	Characterizing sets	23
3.7	Reduction over ATCs	23
3.8	Reduction over ATCs applied after input sequences	23
4	The lower bound function	25
4.1	Permutation function Perm	25
4.2	Helper predicates	26
4.3	Function R	26
4.4	Function RP	28
4.5	Conditions for the result of LB to be a valid lower bound	29
4.6	Function LB	31
4.7	Validity of the result of LB constituting a lower bound	33

5	Test suite generated by the Adaptive State Counting Algorithm	34
5.1	Maximum length contained prefix	34
5.2	Function N	35
5.3	Functions TS, C, RM	35
5.4	Basic properties of the test suite calculation functions	36
5.5	Final iteration	38
6	Sufficiency of the test suite to test for reduction	39
6.1	Properties of minimal sequences to failures extending the deterministic state cover	39
6.2	Sufficiency of the test suite to test for reduction	40
6.3	Main result	40
7	Correctness of the Adaptive State Counting Algorithm in Hoare-Logic	41
8	Example product machines and properties	42
8.1	Constructing FSMs from transition relations	42
8.2	Example FSMs and properties	43

```

theory FSM
imports
  Transition-Systems-and-Automata.Sequence-Zip
  Transition-Systems-and-Automata.Transition-System
  Transition-Systems-and-Automata.Transition-System-Extra
  Transition-Systems-and-Automata.Transition-System-Construction
begin

```

1 Finite state machines

We formalise finite state machines as a 4-tuples, omitting the explicit formulation of the state set, as it can easily be calculated from the successor function. This definition does not require the successor function to be restricted to the input or output alphabet, which is later expressed by the property `well_formed`, together with the finiteness of the state set.

```

record ('in, 'out, 'state) FSM =
  succ :: ('in × 'out) ⇒ 'state ⇒ 'state set
  inputs :: 'in set
  outputs :: 'out set
  initial :: 'state

```

1.1 FSMs as transition systems

We interpret FSMs as transition systems with a singleton initial state set, based on [1].

```

global-interpretation FSM : transition-system-initial
  λ a p. snd a — execute
  λ a p. snd a ∈ succ A (fst a) p — enabled
  λ p. p = initial A — initial
for A
defines path = FSM.path
  and run = FSM.run
  and reachable = FSM.reachable
  and nodes = FSM.nodes
  ⟨proof⟩

```

```

abbreviation size-FSM M ≡ card (nodes M)

```

```

notation
  size-FSM (⟨(|-)|⟩)

```

1.2 Language

The following definitions establish basic notions for FSMs similarly to those of nondeterministic finite automata as defined in [1].

In particular, the language of an FSM state are the IO-parts of the paths in the FSM enabled from that state.

```

abbreviation target ≡ FSM.target
abbreviation states ≡ FSM.states

```

abbreviation $trace \equiv FSM.trace$

abbreviation $successors :: ('in, 'out, 'state, 'more) FSM-scheme \Rightarrow 'state \Rightarrow 'state\ set$ **where**
 $successors \equiv FSM.successors\ TYPE('in)\ TYPE('out)\ TYPE('more)$

lemma $states-alt-def: states\ r\ p = map\ snd\ r$
 $\langle proof \rangle$

lemma $trace-alt-def: trace\ r\ p = smap\ snd\ r$
 $\langle proof \rangle$

definition $language-state :: ('in, 'out, 'state) FSM \Rightarrow 'state$
 $\Rightarrow ('in \times 'out)\ list\ set\ (\langle LS \rangle)$

where

$language-state\ M\ q \equiv \{map\ fst\ r \mid r . path\ M\ r\ q\}$

The language of an FSM is the language of its initial state.

abbreviation $L\ M \equiv LS\ M\ (initial\ M)$

lemma $language-state-alt-def : LS\ M\ q = \{io \mid io\ tr . path\ M\ (io \parallel tr)\ q \wedge length\ io = length\ tr\}$
 $\langle proof \rangle$

lemma $language-state[intro]:$
assumes $path\ M\ (w \parallel r)\ q\ length\ w = length\ r$
shows $w \in LS\ M\ q$
 $\langle proof \rangle$

lemma $language-state-elim[elim]:$
assumes $w \in LS\ M\ q$
obtains r
where $path\ M\ (w \parallel r)\ q\ length\ w = length\ r$
 $\langle proof \rangle$

lemma $language-state-split:$
assumes $w1\ @\ w2 \in LS\ M\ q$
obtains $tr1\ tr2$
where $path\ M\ (w1 \parallel tr1)\ q\ length\ w1 = length\ tr1$
 $path\ M\ (w2 \parallel tr2)\ (target\ (w1 \parallel tr1)\ q)\ length\ w2 = length\ tr2$
 $\langle proof \rangle$

lemma $language-state-prefix :$
assumes $w1\ @\ w2 \in LS\ M\ q$
shows $w1 \in LS\ M\ q$
 $\langle proof \rangle$

lemma $succ-nodes :$
fixes $A :: ('a, 'b, 'c) FSM$
and $w :: ('a \times 'b)$
assumes $q2 \in succ\ A\ w\ q1$
and $q1 \in nodes\ A$
shows $q2 \in nodes\ A$
 $\langle proof \rangle$

lemma $states-target-index :$
assumes $i < length\ p$
shows $(states\ p\ q1) ! i = target\ (take\ (Suc\ i)\ p)\ q1$
 $\langle proof \rangle$

1.3 Product machine for language intersection

The following describes the construction of a product machine from two FSMs M1 and M2 such that the language of the product machine is the intersection of the language of M1 and the language of M2.

definition *product* :: ('in, 'out, 'state1) FSM \Rightarrow ('in, 'out, 'state2) FSM \Rightarrow
('in, 'out, 'state1 \times 'state2) FSM **where**
product A B \equiv
(|
succ = λ a (p₁, p₂). succ A a p₁ \times succ B a p₂,
inputs = inputs A \cup inputs B,
outputs = outputs A \cup outputs B,
initial = (initial A, initial B)
|)

lemma *product-simps*[simp]:
succ (product A B) a (p₁, p₂) = succ A a p₁ \times succ B a p₂
inputs (product A B) = inputs A \cup inputs B
outputs (product A B) = outputs A \cup outputs B
initial (product A B) = (initial A, initial B)
⟨proof⟩

lemma *product-target*[simp]:
assumes length w = length r₁ length r₁ = length r₂
shows target (w || r₁ || r₂) (p₁, p₂) = (target (w || r₁) p₁, target (w || r₂) p₂)
⟨proof⟩

lemma *product-path*[iff]:
assumes length w = length r₁ length r₁ = length r₂
shows path (product A B) (w || r₁ || r₂) (p₁, p₂) \longleftrightarrow path A (w || r₁) p₁ \wedge path B (w || r₂) p₂
⟨proof⟩

lemma *product-language-state*[simp]: LS (product A B) (q₁, q₂) = LS A q₁ \cap LS B q₂
⟨proof⟩

lemma *product-nodes* :
nodes (product A B) \subseteq nodes A \times nodes B
⟨proof⟩

1.4 Required properties

FSMs used by the adaptive state counting algorithm are required to satisfy certain properties which are introduced in here. Most notably, the observability property (see function **observable**) implies the uniqueness of certain paths and hence allows for several stronger variations of previous results.

fun *finite-FSM* :: ('in, 'out, 'state) FSM \Rightarrow bool **where**
finite-FSM M = (finite (nodes M)
 \wedge finite (inputs M)
 \wedge finite (outputs M))

fun *observable* :: ('in, 'out, 'state) FSM \Rightarrow bool **where**
observable M = (\forall t . \forall s1 . ((succ M) t s1 = {})
 \vee (\exists s2 . (succ M) t s1 = {s2}))

fun *completely-specified* :: ('in, 'out, 'state) FSM \Rightarrow bool **where**
completely-specified M = (\forall s1 \in nodes M . \forall x \in inputs M .
 \exists y \in outputs M .
 \exists s2 . s2 \in (succ M) (x,y) s1)

fun *well-formed* :: ('in, 'out, 'state) FSM \Rightarrow bool **where**
well-formed M = (finite-FSM M
 \wedge (\forall s1 x y . (x \notin inputs M \vee y \notin outputs M)
 \longrightarrow succ M (x,y) s1 = {})
 \wedge inputs M \neq {}
 \wedge outputs M \neq {})

abbreviation *OFSM* M \equiv well-formed M
 \wedge observable M
 \wedge completely-specified M

lemma *OFSM-props*[elim!]:
assumes OFSM M

shows *well-formed* M
observable M
completely-specified M \langle *proof* \rangle

lemma *set-of-succs-finite* :
assumes *well-formed* M
and $q \in \text{nodes } M$
shows *finite* (*succ* M io q)
 \langle *proof* \rangle

lemma *well-formed-path-io-containment* :
assumes *well-formed* M
and $\text{path } M$ p q
shows $\text{set } (\text{map } \text{fst } p) \subseteq (\text{inputs } M \times \text{outputs } M)$
 \langle *proof* \rangle

lemma *path-input-containment* :
assumes *well-formed* M
and $\text{path } M$ p q
shows $\text{set } (\text{map } \text{fst } (\text{map } \text{fst } p)) \subseteq \text{inputs } M$
 \langle *proof* \rangle

lemma *path-state-containment* :
assumes $\text{path } M$ p q
and $q \in \text{nodes } M$
shows $\text{set } (\text{map } \text{snd } p) \subseteq \text{nodes } M$
 \langle *proof* \rangle

lemma *language-state-inputs* :
assumes *well-formed* M
and $io \in \text{language-state } M$ q
shows $\text{set } (\text{map } \text{fst } io) \subseteq \text{inputs } M$
 \langle *proof* \rangle

lemma *set-of-paths-finite* :
assumes *well-formed* M
and $q1 \in \text{nodes } M$
shows *finite* $\{ p . \text{path } M$ p $q1 \wedge \text{target } p$ $q1 = q2 \wedge \text{length } p \leq k \}$
 \langle *proof* \rangle

lemma *non-distinct-duplicate-indices* :
assumes $\neg \text{distinct } xs$
shows $\exists i1\ i2 . i1 \neq i2 \wedge xs ! i1 = xs ! i2 \wedge i1 \leq \text{length } xs \wedge i2 \leq \text{length } xs$
 \langle *proof* \rangle

lemma *reaching-path-without-repetition* :
assumes *well-formed* M
and $q2 \in \text{reachable } M$ $q1$
and $q1 \in \text{nodes } M$
shows $\exists p . \text{path } M$ p $q1 \wedge \text{target } p$ $q1 = q2 \wedge \text{distinct } (q1 \# \text{states } p$ $q1)$
 \langle *proof* \rangle

lemma *observable-path-unique*[*simp*] :
assumes $io \in LS$ M q
and *observable* M
and $\text{path } M$ ($io \parallel tr1$) q $\text{length } io = \text{length } tr1$
and $\text{path } M$ ($io \parallel tr2$) q $\text{length } io = \text{length } tr2$
shows $tr1 = tr2$

$\langle proof \rangle$

lemma *observable-path-unique-ex*[*elim*] :
 assumes *observable M*
 and $io \in LS\ M\ q$
obtains *tr*
where $\{ t . path\ M\ (io\ ||\ t)\ q \wedge length\ io = length\ t \} = \{ tr \}$
 $\langle proof \rangle$

lemma *well-formed-product*[*simp*] :
 assumes *well-formed M1*
 and *well-formed M2*
shows *well-formed (product M2 M1) (is well-formed ?PM)*
 $\langle proof \rangle$

1.5 States reached by a given IO-sequence

Function `io_targets` collects all states of an FSM reached from a given state by a given IO-sequence. Notably, for any observable FSM, this set contains at most one state.

fun *io-targets* :: (*'in, 'out, 'state*) *FSM* \Rightarrow *'state* \Rightarrow (*'in* \times *'out*) *list* \Rightarrow *'state set* **where**
 io-targets M q io = $\{ target\ (io\ ||\ tr)\ q \mid tr . path\ M\ (io\ ||\ tr)\ q \wedge length\ io = length\ tr \}$

lemma *io-target-implies-L* :
 assumes $q \in io_targets\ M\ (initial\ M)\ io$
 shows $io \in L\ M$
 $\langle proof \rangle$

lemma *io-target-from-path* :
 assumes *path M (w || tr) q*
 and $length\ w = length\ tr$
shows $target\ (w\ ||\ tr)\ q \in io_targets\ M\ q\ w$
 $\langle proof \rangle$

lemma *io-targets-observable-singleton-ex* :
 assumes *observable M*
 and $io \in LS\ M\ q1$
shows $\exists\ q2 . io_targets\ M\ q1\ io = \{ q2 \}$
 $\langle proof \rangle$

lemma *io-targets-observable-singleton-ob* :
 assumes *observable M*
 and $io \in LS\ M\ q1$
obtains *q2*
 where $io_targets\ M\ q1\ io = \{ q2 \}$
 $\langle proof \rangle$

lemma *io-targets-elim*[*elim*] :
 assumes $p \in io_targets\ M\ q\ io$
obtains *tr*
where $target\ (io\ ||\ tr)\ q = p \wedge path\ M\ (io\ ||\ tr)\ q \wedge length\ io = length\ tr$
 $\langle proof \rangle$

lemma *io-targets-reachable* :
 assumes $q2 \in io_targets\ M\ q1\ io$
 shows $q2 \in reachable\ M\ q1$
 $\langle proof \rangle$

lemma *io-targets-nodes* :
 assumes $q2 \in io_targets\ M\ q1\ io$
 and $q1 \in nodes\ M$
shows $q2 \in nodes\ M$
 $\langle proof \rangle$

lemma *observable-io-targets-split* :
assumes *observable M*
and *io-targets M q1 (vs @ xs) = {q3}*
and *io-targets M q1 vs = {q2}*
shows *io-targets M q2 xs = {q3}*
<proof>

lemma *observable-io-target-unique-target* :
assumes *observable M*
and *io-targets M q1 io = {q2}*
and *path M (io || tr) q1*
and *length io = length tr*
shows *target (io || tr) q1 = q2*
<proof>

lemma *target-in-states* :
assumes *length io = length tr*
and *length io > 0*
shows *last (states (io || tr) q) = target (io || tr) q*
<proof>

lemma *target-alt-def* :
assumes *length io = length tr*
shows *length io = 0 \implies target (io || tr) q = q*
length io > 0 \implies target (io || tr) q = last tr
<proof>

lemma *obs-target-is-io-targets* :
assumes *observable M*
and *path M (io || tr) q*
and *length io = length tr*
shows *io-targets M q io = {target (io || tr) q}*
<proof>

lemma *io-target-target* :
assumes *io-targets M q1 io = {q2}*
and *path M (io || tr) q1*
and *length io = length tr*
shows *target (io || tr) q1 = q2*
<proof>

lemma *index-last-take* :
assumes *i < length xs*
shows *xs ! i = last (take (Suc i) xs)*
<proof>

lemma *path-last-io-target* :
assumes *path M (xs || tr) q*
and *length xs = length tr*
and *length xs > 0*
shows *last tr \in io-targets M q xs*
<proof>

lemma *path-prefix-io-targets* :
assumes *path M (xs || tr) q*
and *length xs = length tr*
and *length xs > 0*
shows *last (take (Suc i) tr) \in io-targets M q (take (Suc i) xs)*
<proof>

lemma *states-index-io-target* :
assumes $i < \text{length } xs$
and $\text{path } M (xs \parallel tr) q$
and $\text{length } xs = \text{length } tr$
and $\text{length } xs > 0$
shows $(\text{states } (xs \parallel tr) q) ! i \in \text{io-targets } M q (\text{take } (Suc\ i) xs)$
 $\langle \text{proof} \rangle$

lemma *observable-io-targets-append* :
assumes *observable* M
and $\text{io-targets } M q1\ vs = \{q2\}$
and $\text{io-targets } M q2\ xs = \{q3\}$
shows $\text{io-targets } M q1\ (vs@xs) = \{q3\}$
 $\langle \text{proof} \rangle$

lemma *io-path-states-prefix* :
assumes *observable* M
and $\text{path } M (io1 \parallel tr1) q$
and $\text{length } tr1 = \text{length } io1$
and $\text{path } M (io2 \parallel tr2) q$
and $\text{length } tr2 = \text{length } io2$
and $\text{prefix } io1\ io2$
shows $tr1 = \text{take } (\text{length } tr1)\ tr2$
 $\langle \text{proof} \rangle$

lemma *observable-io-targets-suffix* :
assumes *observable* M
and $\text{io-targets } M q1\ vs = \{q2\}$
and $\text{io-targets } M q1\ (vs@xs) = \{q3\}$
shows $\text{io-targets } M q2\ xs = \{q3\}$
 $\langle \text{proof} \rangle$

lemma *observable-io-target-is-singleton[simp]* :
assumes *observable* M
and $p \in \text{io-targets } M q\ io$
shows $\text{io-targets } M q\ io = \{p\}$
 $\langle \text{proof} \rangle$

lemma *observable-path-prefix* :
assumes *observable* M
and $\text{path } M (io \parallel tr) q$
and $\text{length } io = \text{length } tr$
and $\text{path } M (ioP \parallel trP) q$
and $\text{length } ioP = \text{length } trP$
and $\text{prefix } ioP\ io$
shows $trP = \text{take } (\text{length } ioP)\ tr$
 $\langle \text{proof} \rangle$

lemma *io-targets-succ* :
assumes $q2 \in \text{io-targets } M q1\ [xy]$
shows $q2 \in \text{succ } M xy\ q1$
 $\langle \text{proof} \rangle$

1.6 D-reachability

A state of some FSM is d-reached (deterministically reached) by some input sequence if any sequence in the language of the FSM with this input sequence reaches that state. That state is then called d-reachable.

abbreviation $d\text{-reached-by } M p xs q tr ys \equiv$
 $((length\ xs = length\ ys \wedge length\ xs = length\ tr$
 $\wedge (path\ M ((xs\ ||\ ys)\ ||\ tr)\ p) \wedge target\ ((xs\ ||\ ys)\ ||\ tr)\ p = q)$
 $\wedge (\forall\ ys2\ tr2 . (length\ xs = length\ ys2 \wedge length\ xs = length\ tr2$
 $\wedge path\ M ((xs\ ||\ ys2)\ ||\ tr2)\ p \longrightarrow target\ ((xs\ ||\ ys2)\ ||\ tr2)\ p = q))$

fun $d\text{-reaches} :: ('in, 'out, 'state) FSM \Rightarrow 'state \Rightarrow 'in\ list \Rightarrow 'state \Rightarrow bool$ **where**
 $d\text{-reaches } M p xs q = (\exists\ tr\ ys . d\text{-reached-by } M p xs q tr ys)$

fun $d\text{-reachable} :: ('in, 'out, 'state) FSM \Rightarrow 'state \Rightarrow 'state\ set$ **where**
 $d\text{-reachable } M p = \{ q . (\exists\ xs . d\text{-reaches } M p xs q) \}$

lemma $d\text{-reaches-unique}[elim] :$

assumes $d\text{-reaches } M p xs q1$

and $d\text{-reaches } M p xs q2$

shows $q1 = q2$

$\langle proof \rangle$

lemma $d\text{-reaches-unique-cases}[simp] : \{ q . d\text{-reaches } M (initial\ M) xs q \} = \{ \}$
 $\vee (\exists\ q2 . \{ q . d\text{-reaches } M (initial\ M) xs q \} = \{ q2 \})$

$\langle proof \rangle$

lemma $d\text{-reaches-unique-obtain}[simp] :$

assumes $d\text{-reaches } M (initial\ M) xs q$

shows $\{ p . d\text{-reaches } M (initial\ M) xs p \} = \{ q \}$

$\langle proof \rangle$

lemma $d\text{-reaches-io-target} :$

assumes $d\text{-reaches } M p xs q$

and $length\ ys = length\ xs$

shows $io\text{-targets } M p (xs\ ||\ ys) \subseteq \{ q \}$

$\langle proof \rangle$

lemma $d\text{-reachable-reachable} : d\text{-reachable } M p \subseteq reachable\ M p$

$\langle proof \rangle$

1.7 Deterministic state cover

The deterministic state cover of some FSM is a minimal set of input sequences such that every d-reachable state of the FSM is d-reached by a sequence in the set and the set contains the empty sequence (which d-reaches the initial state).

fun $is\text{-det-state-cover-ass} :: ('in, 'out, 'state) FSM \Rightarrow ('state \Rightarrow 'in\ list) \Rightarrow bool$ **where**
 $is\text{-det-state-cover-ass } M f = (f (initial\ M) = [] \wedge (\forall\ s \in d\text{-reachable } M (initial\ M) .$
 $d\text{-reaches } M (initial\ M) (f\ s)\ s))$

lemma $det\text{-state-cover-ass-dist} :$

assumes $is\text{-det-state-cover-ass } M f$

and $s1 \in d\text{-reachable } M (initial\ M)$

and $s2 \in d\text{-reachable } M (initial\ M)$

and $s1 \neq s2$

shows $\neg(d\text{-reaches } M (initial\ M) (f\ s2)\ s1)$

$\langle proof \rangle$

lemma $det\text{-state-cover-ass-diff} :$

assumes $is\text{-det-state-cover-ass } M f$

and $s1 \in d\text{-reachable } M (initial\ M)$

and $s2 \in d\text{-reachable } M (initial\ M)$

and $s1 \neq s2$

shows $f\ s1 \neq f\ s2$

$\langle proof \rangle$

fun $is\text{-det-state-cover} :: ('in, 'out, 'state) FSM \Rightarrow 'in\ list\ set \Rightarrow bool$ **where**

$$is-det-state-cover M V = (\exists f . is-det-state-cover-ass M f \\ \wedge V = image f (d-reachable M (initial M)))$$

lemma *det-state-cover-d-reachable*[*elim*] :
assumes *is-det-state-cover* *M V*
and $v \in V$
obtains *q*
where *d-reaches* *M (initial M) v q*
 $\langle proof \rangle$

lemma *det-state-cover-card*[*simp*] :
assumes *is-det-state-cover* *M V*
and *finite (nodes M)*
shows $card (d-reachable M (initial M)) = card V$
 $\langle proof \rangle$

lemma *det-state-cover-finite* :
assumes *is-det-state-cover* *M V*
and *finite (nodes M)*
shows *finite V*
 $\langle proof \rangle$

lemma *det-state-cover-initial* :
assumes *is-det-state-cover* *M V*
shows $\square \in V$
 $\langle proof \rangle$

lemma *det-state-cover-empty* :
assumes *is-det-state-cover* *M V*
shows $\square \in V$
 $\langle proof \rangle$

1.8 IO reduction

An FSM is a reduction of another, if its language is a subset of the language of the latter FSM.

fun *io-reduction* :: $('in, 'out, 'state) FSM \Rightarrow ('in, 'out, 'state) FSM \\ \Rightarrow bool$ (**infix** $\langle \preceq \rangle$ 200)
where
 $M1 \preceq M2 = (LS M1 (initial M1) \subseteq LS M2 (initial M2))$

lemma *language-state-inclusion-of-state-reached-by-same-sequence* :
assumes $LS M1 q1 \subseteq LS M2 q2$
and *observable M1*
and *observable M2*
and $io-targets M1 q1 io = \{ q1t \}$
and $io-targets M2 q2 io = \{ q2t \}$
shows $LS M1 q1t \subseteq LS M2 q2t$
 $\langle proof \rangle$

1.9 Language subsets for input sequences

The following definitions describe restrictions of languages to only those IO-sequences that exhibit a certain input sequence or whose input sequence is contained in a given set of input sequences. This allows to define the notion that some FSM is a reduction of another over a given set of input sequences, but not necessarily over the entire language of the latter FSM.

fun *language-state-for-input* ::
 $('in, 'out, 'state) FSM \Rightarrow 'state \Rightarrow 'in list \Rightarrow ('in \times 'out) list set$ **where**
 $language-state-for-input M q xs = \{(xs \parallel ys) \mid ys . (length xs = length ys \wedge (xs \parallel ys) \in LS M q)\}$

fun *language-state-for-inputs* ::
 $('in, 'out, 'state) FSM \Rightarrow 'state \Rightarrow 'in list set \Rightarrow ('in \times 'out) list set$

$\langle (LS_{in} \text{ -- } -) \rangle [1000,1000,1000]$ **where**
language-state-for-inputs $M \ q \ ISeqs = \{(xs \parallel ys) \mid xs \ ys . (xs \in ISeqs$
 $\wedge \text{length } xs = \text{length } ys$
 $\wedge (xs \parallel ys) \in LS \ M \ q)\}$

abbreviation $L_{in} \ M \ TS \equiv LS_{in} \ M \ (\text{initial } M) \ TS$

abbreviation *io-reduction-on* $M1 \ TS \ M2 \equiv (L_{in} \ M1 \ TS \subseteq L_{in} \ M2 \ TS)$

notation

io-reduction-on $\langle (- \preceq [-] -) \rangle [1000,0,0] \ 61$

notation (*latex output*)

io-reduction-on $\langle (- \preceq -) \rangle [1000,0,0] \ 61$

lemma *language-state-for-input-alt-def* :

language-state-for-input $M \ q \ xs = LS_{in} \ M \ q \ \{xs\}$

$\langle \text{proof} \rangle$

lemma *language-state-for-inputs-alt-def* :

$LS_{in} \ M \ q \ ISeqs = \bigcup (\text{image } (\text{language-state-for-input } M \ q) \ ISeqs)$

$\langle \text{proof} \rangle$

lemma *language-state-for-inputs-in-language-state* :

$LS_{in} \ M \ q \ T \subseteq \text{language-state } M \ q$

$\langle \text{proof} \rangle$

lemma *language-state-for-inputs-map-fst* :

assumes $io \in \text{language-state } M \ q$

and $\text{map } fst \ io \in T$

shows $io \in LS_{in} \ M \ q \ T$

$\langle \text{proof} \rangle$

lemma *language-state-for-inputs-nonempty* :

assumes $set \ xs \subseteq \text{inputs } M$

and $\text{completely-specified } M$

and $q \in \text{nodes } M$

shows $LS_{in} \ M \ q \ \{xs\} \neq \{\}$

$\langle \text{proof} \rangle$

lemma *language-state-for-inputs-map-fst-contained* :

assumes $vs \in LS_{in} \ M \ q \ V$

shows $\text{map } fst \ vs \in V$

$\langle \text{proof} \rangle$

lemma *language-state-for-inputs-empty* :

assumes $\square \in V$

shows $\square \in LS_{in} \ M \ q \ V$

$\langle \text{proof} \rangle$

lemma *language-state-for-input-empty[simp]* :

language-state-for-input $M \ q \ \square = \{\square\}$

$\langle \text{proof} \rangle$

lemma *language-state-for-input-take* :

assumes $io \in \text{language-state-for-input } M \ q \ xs$

shows $\text{take } n \ io \in \text{language-state-for-input } M \ q \ (\text{take } n \ xs)$

$\langle \text{proof} \rangle$

lemma *language-state-for-inputs-prefix* :

assumes $vs@xs \in L_{in} \ M1 \ \{vs'@xs'\}$

and $\text{length } vs = \text{length } vs'$

shows $vs \in L_{in} \ M1 \ \{vs'\}$

$\langle \text{proof} \rangle$

lemma *language-state-for-inputs-union* :

shows $LS_{in} M q T1 \cup LS_{in} M q T2 = LS_{in} M q (T1 \cup T2)$
 ⟨proof⟩

lemma *io-reduction-on-subset* :
assumes *io-reduction-on M1 T M2*
and $T' \subseteq T$
shows *io-reduction-on M1 T' M2*
 ⟨proof⟩

1.10 Sequences to failures

A sequence to a failure for FSMs M1 and M2 is a sequence such that any proper prefix of it is contained in the languages of both M1 and M2, while the sequence itself is contained only in the language of A.

That is, if a sequence to a failure for M1 and M2 exists, then M1 is not a reduction of M2.

fun *sequence-to-failure* ::
 (*'in,'out,'state*) FSM \Rightarrow (*'in,'out,'state*) FSM \Rightarrow (*'in* \times *'out*) list \Rightarrow bool **where**
sequence-to-failure M1 M2 xs = (
 (butlast xs) \in (language-state M2 (initial M2) \cap language-state M1 (initial M1))
 \wedge xs \in (language-state M1 (initial M1) $-$ language-state M2 (initial M2)))

lemma *sequence-to-failure-ob* :
assumes $\neg M1 \preceq M2$
and *well-formed M1*
and *well-formed M2*
obtains *io*
where *sequence-to-failure M1 M2 io*
 ⟨proof⟩

lemma *sequence-to-failure-succ* :
assumes *sequence-to-failure M1 M2 io*
shows $\forall q \in io\text{-targets } M2 \text{ (initial } M2) \text{ (butlast } io) . succ } M2 \text{ (last } io) q = \{\}$
 ⟨proof⟩

lemma *sequence-to-failure-non-nil* :
assumes *sequence-to-failure M1 M2 xs*
shows $xs \neq []$
 ⟨proof⟩

lemma *sequence-to-failure-from-arbitrary-failure* :
assumes $vs@xs \in L M1 - L M2$
and $vs \in L M2 \cap L M1$
shows $\exists xs' . prefix } xs' xs \wedge sequence-to-failure } M1 M2 (vs@xs')$
 ⟨proof⟩

The following lemma shows that if M1 is not a reduction of M2, then a minimal sequence to a failure exists that is of length at most the number of states in M1 times the number of states in M2.

lemma *sequence-to-failure-length* :
assumes *well-formed M1*
and *well-formed M2*
and *observable M1*
and *observable M2*
and $\neg M1 \preceq M2$
shows $\exists xs . sequence-to-failure } M1 M2 xs \wedge length } xs \leq |M2| * |M1|$
 ⟨proof⟩

1.11 Minimal sequence to failure extending

A minimal sequence to a failure extending some some set of IO-sequences is a sequence to a failure of minimal length such that a prefix of that sequence is contained in the set.

fun *minimal-sequence-to-failure-extending* ::
 (*'in list set*) \Rightarrow (*'in,'out,'state*) FSM \Rightarrow (*'in,'out,'state*) FSM \Rightarrow (*'in* \times *'out*) list
 \Rightarrow (*'in* \times *'out*) list \Rightarrow bool **where**
minimal-sequence-to-failure-extending V M1 M2 v' io = (

$$v' \in L_{in} M1 V \wedge \text{sequence-to-failure } M1 M2 (v' @ io) \\ \wedge \neg (\exists io' . \exists w' \in L_{in} M1 V . \text{sequence-to-failure } M1 M2 (w' @ io') \\ \wedge \text{length } io' < \text{length } io)$$

lemma *minimal-sequence-to-failure-extending-det-state-cover-ob* :

assumes *well-formed M1*
and *well-formed M2*
and *observable M2*
and *is-det-state-cover M2 V*
and $\neg M1 \preceq M2$

obtains *vs xs*

where *minimal-sequence-to-failure-extending V M1 M2 vs xs*

<proof>

lemma *mstfe-prefix-input-in-V* :

assumes *minimal-sequence-to-failure-extending V M1 M2 vs xs*
shows $(\text{map fst } vs) \in V$

<proof>

1.12 Complete test suite derived from the product machine

The classical result of testing FSMs for language inclusion : Any failure can be observed by a sequence of length at most $n*m$ where n is the number of states of the reference model (here FSM $M2$) and m is an upper bound on the number of states of the SUT (here FSM $M1$).

lemma *product-suite-soundness* :

assumes *well-formed M1*
and *well-formed M2*
and *observable M1*
and *observable M2*
and $\text{inputs } M2 = \text{inputs } M1$
and $|M1| \leq m$

shows $\neg M1 \preceq M2 \longrightarrow \neg M1 \preceq \llbracket \{xs . \text{set } xs \subseteq \text{inputs } M2 \wedge \text{length } xs \leq |M2| * m\} \rrbracket M2$
(is $\neg M1 \preceq M2 \longrightarrow \neg M1 \preceq \llbracket ?TS \rrbracket M2$)

<proof>

lemma *product-suite-completeness* :

assumes *well-formed M1*
and *well-formed M2*
and *observable M1*
and *observable M2*
and $\text{inputs } M2 = \text{inputs } M1$
and $|M1| \leq m$

shows $M1 \preceq M2 \longleftrightarrow M1 \preceq \llbracket \{xs . \text{set } xs \subseteq \text{inputs } M2 \wedge \text{length } xs \leq |M2| * m\} \rrbracket M2$
(is $M1 \preceq M2 \longleftrightarrow M1 \preceq \llbracket ?TS \rrbracket M2$)

<proof>

end

theory *FSM-Product*

imports *FSM*

begin

2 Product machines with an additional fail state

We extend the product machine for language intersection presented in theory *FSM* by an additional state that is reached only by sequences such that any proper prefix of the sequence is in the language intersection, whereas the full sequence is only contained in the language of the machine B for which we want to check whether it is a reduction of some machine A .

To allow for free choice of the FAIL state, we define the following property that holds iff AB is the product machine of A and B extended with fail state *FAIL*.

fun *productF* :: $(\text{'in}, \text{'out}, \text{'state1}) \text{FSM} \Rightarrow (\text{'in}, \text{'out}, \text{'state2}) \text{FSM} \Rightarrow (\text{'state1} \times \text{'state2})$
 $\Rightarrow (\text{'in}, \text{'out}, \text{'state1} \times \text{'state2}) \text{FSM} \Rightarrow \text{bool}$ **where**

```

productF A B FAIL AB = (
  (inputs A = inputs B)
  ∧ (fst FAIL ∉ nodes A)
  ∧ (snd FAIL ∉ nodes B)
  ∧ AB = (
    succ = (λ a (p1,p2) . (if (p1 ∈ nodes A ∧ p2 ∈ nodes B ∧ (fst a ∈ inputs A)
      ∧ (snd a ∈ outputs A ∪ outputs B))
      then (if (succ A a p1 = {} ∧ succ B a p2 ≠ {})
        then {FAIL}
        else (succ A a p1 × succ B a p2))
      else {})),
    inputs = inputs A,
    outputs = outputs A ∪ outputs B,
    initial = (initial A, initial B)
  )
)

```

lemma *productF-simps[simp]*:

```

productF A B FAIL AB ⇒ succ AB a (p1,p2) = (if (p1 ∈ nodes A ∧ p2 ∈ nodes B
  ∧ (fst a ∈ inputs A) ∧ (snd a ∈ outputs A ∪ outputs B))
  then (if (succ A a p1 = {} ∧ succ B a p2 ≠ {})
    then {FAIL}
    else (succ A a p1 × succ B a p2))
  else {})

productF A B FAIL AB ⇒ inputs AB = inputs A
productF A B FAIL AB ⇒ outputs AB = outputs A ∪ outputs B
productF A B FAIL AB ⇒ initial AB = (initial A, initial B)
⟨proof⟩

```

lemma *fail-next-productF* :

```

assumes well-formed M1
and well-formed M2
and productF M2 M1 FAIL PM
shows succ PM a FAIL = {}
⟨proof⟩

```

lemma *nodes-productF* :

```

assumes well-formed M1
and well-formed M2
and productF M2 M1 FAIL PM
shows nodes PM ⊆ insert FAIL (nodes M2 × nodes M1)
⟨proof⟩

```

lemma *well-formed-productF[simp]* :

```

assumes well-formed M1
and well-formed M2
and productF M2 M1 FAIL PM
shows well-formed PM
⟨proof⟩

```

lemma *observable-productF[simp]* :

```

assumes observable M1
and observable M2
and productF M2 M1 FAIL PM
shows observable PM
⟨proof⟩

```

lemma *no-transition-after-FAIL* :

```

assumes productF A B FAIL AB

```

shows $\text{succ } AB \text{ io } FAIL = \{\}$
 ⟨proof⟩

lemma *no-prefix-targets-FAIL* :
assumes $\text{productF } M2 \ M1 \ FAIL \ PM$
and $\text{path } PM \ p \ q$
and $k < \text{length } p$
shows $\text{target } (\text{take } k \ p) \ q \neq FAIL$
 ⟨proof⟩

lemma *productF-path-inclusion* :
assumes $\text{length } w = \text{length } r1 \ \text{length } r1 = \text{length } r2$
and $\text{productF } A \ B \ FAIL \ AB$
and $\text{well-formed } A$
and $\text{well-formed } B$
and $\text{path } A \ (w \ || \ r1) \ p1 \wedge \text{path } B \ (w \ || \ r2) \ p2$
and $p1 \in \text{nodes } A$
and $p2 \in \text{nodes } B$
shows $\text{path } (AB) \ (w \ || \ r1 \ || \ r2) \ (p1, \ p2)$
 ⟨proof⟩

lemma *productF-path-forward* :
assumes $\text{length } w = \text{length } r1 \ \text{length } r1 = \text{length } r2$
and $\text{productF } A \ B \ FAIL \ AB$
and $\text{well-formed } A$
and $\text{well-formed } B$
and $(\text{path } A \ (w \ || \ r1) \ p1 \wedge \text{path } B \ (w \ || \ r2) \ p2)$
 $\vee (\text{target } (w \ || \ r1 \ || \ r2) \ (p1, \ p2) = FAIL$
 $\wedge \text{length } w > 0$
 $\wedge \text{path } A \ (\text{butlast } (w \ || \ r1)) \ p1$
 $\wedge \text{path } B \ (\text{butlast } (w \ || \ r2)) \ p2$
 $\wedge \text{succ } A \ (\text{last } w) \ (\text{target } (\text{butlast } (w \ || \ r1)) \ p1) = \{\}$
 $\wedge \text{succ } B \ (\text{last } w) \ (\text{target } (\text{butlast } (w \ || \ r2)) \ p2) \neq \{\})$
and $p1 \in \text{nodes } A$
and $p2 \in \text{nodes } B$
shows $\text{path } (AB) \ (w \ || \ r1 \ || \ r2) \ (p1, \ p2)$
 ⟨proof⟩

lemma *butlast-zip-cons* : $\text{length } ws = \text{length } r1s \implies ws \neq []$
 $\implies \text{butlast } (w \ \# \ ws \ || \ r1 \ \# \ r1s) = ((w, r1) \ \# \ (\text{butlast } (ws \ || \ r1s)))$
 ⟨proof⟩

lemma *productF-succ-fail-imp* :
assumes $\text{productF } A \ B \ FAIL \ AB$
and $FAIL \in \text{succ } AB \ w \ (p1, p2)$
and $\text{well-formed } A$
and $\text{well-formed } B$
shows $p1 \in \text{nodes } A \wedge p2 \in \text{nodes } B \wedge (\text{fst } w \in \text{inputs } A) \wedge (\text{snd } w \in \text{outputs } A \cup \text{outputs } B)$
 $\wedge \text{succ } AB \ w \ (p1, p2) = \{FAIL\} \wedge \text{succ } A \ w \ p1 = \{\} \wedge \text{succ } B \ w \ p2 \neq \{\}$
 ⟨proof⟩

lemma *productF-path-reverse* :
assumes $\text{length } w = \text{length } r1 \ \text{length } r1 = \text{length } r2$
and $\text{productF } A \ B \ FAIL \ AB$
and $\text{well-formed } A$
and $\text{well-formed } B$
and $\text{path } AB \ (w \ || \ r1 \ || \ r2) \ (p1, \ p2)$
and $p1 \in \text{nodes } A$

and $p2 \in \text{nodes } B$
shows $(\text{path } A (w \parallel r1) p1 \wedge \text{path } B (w \parallel r2) p2)$
 $\vee (\text{target } (w \parallel r1 \parallel r2) (p1, p2) = \text{FAIL})$
 $\wedge \text{length } w > 0$
 $\wedge \text{path } A (\text{butlast } (w \parallel r1)) p1$
 $\wedge \text{path } B (\text{butlast } (w \parallel r2)) p2$
 $\wedge \text{succ } A (\text{last } w) (\text{target } (\text{butlast } (w \parallel r1)) p1) = \{\}$
 $\wedge \text{succ } B (\text{last } w) (\text{target } (\text{butlast } (w \parallel r2)) p2) \neq \{\}$
 $\langle \text{proof} \rangle$

lemma *butlast- zip [simp]* :
assumes $\text{length } xs = \text{length } ys$
shows $\text{butlast } (xs \parallel ys) = (\text{butlast } xs \parallel \text{butlast } ys)$
 $\langle \text{proof} \rangle$

lemma *productF-path-reverse-ob* :
assumes $\text{length } w = \text{length } r1 \text{ length } r1 = \text{length } r2$
and $\text{productF } A B \text{ FAIL } AB$
and $\text{well-formed } A$
and $\text{well-formed } B$
and $\text{path } AB (w \parallel r1 \parallel r2) (p1, p2)$
and $p1 \in \text{nodes } A$
and $p2 \in \text{nodes } B$
obtains $r2'$
where $\text{path } B (w \parallel r2') p2 \wedge \text{length } w = \text{length } r2'$
 $\langle \text{proof} \rangle$

The following lemma formalizes the property of paths of the product machine as described in the section introduction.

lemma *productF-path[i ff]* :
assumes $\text{length } w = \text{length } r1 \text{ length } r1 = \text{length } r2$
and $\text{productF } A B \text{ FAIL } AB$
and $\text{well-formed } A$
and $\text{well-formed } B$
and $p1 \in \text{nodes } A$
and $p2 \in \text{nodes } B$
shows $\text{path } AB (w \parallel r1 \parallel r2) (p1, p2) \longleftrightarrow ((\text{path } A (w \parallel r1) p1 \wedge \text{path } B (w \parallel r2) p2)$
 $\vee (\text{target } (w \parallel r1 \parallel r2) (p1, p2) = \text{FAIL})$
 $\wedge \text{length } w > 0$
 $\wedge \text{path } A (\text{butlast } (w \parallel r1)) p1$
 $\wedge \text{path } B (\text{butlast } (w \parallel r2)) p2$
 $\wedge \text{succ } A (\text{last } w) (\text{target } (\text{butlast } (w \parallel r1)) p1) = \{\}$
 $\wedge \text{succ } B (\text{last } w) (\text{target } (\text{butlast } (w \parallel r2)) p2) \neq \{\})$ **(is ?path \longleftrightarrow ?paths)
 $\langle \text{proof} \rangle$**

lemma *path-last-succ* :
assumes $\text{path } A (ws \parallel r1s) p1$
and $\text{length } r1s = \text{length } ws$
and $\text{length } ws > 0$
shows $\text{last } r1s \in \text{succ } A (\text{last } ws) (\text{target } (\text{butlast } (ws \parallel r1s)) p1)$
 $\langle \text{proof} \rangle$

lemma *zip-last* :
assumes $\text{length } r1 > 0$
and $\text{length } r1 = \text{length } r2$
shows $\text{last } (r1 \parallel r2) = (\text{last } r1, \text{last } r2)$
 $\langle \text{proof} \rangle$

lemma *productF-path-reverse-ob-2* :
assumes $\text{length } w = \text{length } r1 \text{ length } r1 = \text{length } r2$
and $\text{productF } A B \text{ FAIL } AB$

```

and    well-formed A
and    well-formed B
and    path AB (w || r1 || r2) (p1, p2)
and    p1 ∈ nodes A
and    p2 ∈ nodes B
and    w ∈ language-state A p1
and    observable A
shows path A (w || r1) p1 ∧ length w = length r1 path B (w || r2) p2 ∧ length w = length r2
      target (w || r1) p1 = fst (target (w || r1 || r2) (p1,p2))
      target (w || r2) p2 = snd (target (w || r1 || r2) (p1,p2))
⟨proof⟩

```

```

lemma productF-path-unzip :
  assumes productF A B FAIL AB
  and    path AB (w || tr) q
  and    length tr = length w
shows path AB (w || (map fst tr || map snd tr)) q
⟨proof⟩

```

```

lemma productF-path-io-targets :
  assumes productF A B FAIL AB
  and    io-targets AB (qA,qB) w = {(pA,pB)}
  and    w ∈ language-state A qA
  and    w ∈ language-state B qB
  and    observable A
  and    observable B
  and    well-formed A
  and    well-formed B
  and    qA ∈ nodes A
  and    qB ∈ nodes B
shows pA ∈ io-targets A qA w pB ∈ io-targets B qB w
⟨proof⟩

```

```

lemma productF-path-io-targets-reverse :
  assumes productF A B FAIL AB
  and    pA ∈ io-targets A qA w
  and    pB ∈ io-targets B qB w
  and    w ∈ language-state A qA
  and    w ∈ language-state B qB
  and    observable A
  and    observable B
  and    well-formed A
  and    well-formed B
  and    qA ∈ nodes A
  and    qB ∈ nodes B
shows io-targets AB (qA,qB) w = {(pA,pB)}
⟨proof⟩

```

2.1 Sequences to failure in the product machine

A sequence to a failure for A and B reaches the fail state of any product machine of A and B with added fail state.

```

lemma fail-reachable-by-sequence-to-failure :
  assumes sequence-to-failure M1 M2 io
  and    well-formed M1
  and    well-formed M2

```

and *productF M2 M1 FAIL PM*
obtains *p*
where *path PM (io||p) (initial PM) ∧ length p = length io ∧ target (io||p) (initial PM) = FAIL*
<proof>

lemma *fail-reachable* :
assumes $\neg M1 \preceq M2$
and *well-formed M1*
and *well-formed M2*
and *productF M2 M1 FAIL PM*
shows *FAIL ∈ reachable PM (initial PM)*
<proof>

lemma *fail-reachable-ob* :
assumes $\neg M1 \preceq M2$
and *well-formed M1*
and *well-formed M2*
and *observable M2*
and *productF M2 M1 FAIL PM*
obtains *p*
where *path PM p (initial PM) target p (initial PM) = FAIL*
<proof>

lemma *fail-reachable-reverse* :
assumes *well-formed M1*
and *well-formed M2*
and *productF M2 M1 FAIL PM*
and *FAIL ∈ reachable PM (initial PM)*
and *observable M2*
shows $\neg M1 \preceq M2$
<proof>

lemma *fail-reachable-iff[iff]* :
assumes *well-formed M1*
and *well-formed M2*
and *productF M2 M1 FAIL PM*
and *observable M2*
shows *FAIL ∈ reachable PM (initial PM) ↔ $\neg M1 \preceq M2$*
<proof>

lemma *reaching-path-length* :
assumes *productF A B FAIL AB*
and *well-formed A*
and *well-formed B*
and *q2 ∈ reachable AB q1*
and *q2 ≠ FAIL*
and *q1 ∈ nodes AB*
shows $\exists p . \text{path } AB \ p \ q1 \wedge \text{target } p \ q1 = q2 \wedge \text{length } p < \text{card } (\text{nodes } A) * \text{card } (\text{nodes } B)$
<proof>

lemma *reaching-path-fail-length* :
assumes *productF A B FAIL AB*
and *well-formed A*
and *well-formed B*
and *q2 ∈ reachable AB q1*
and *q1 ∈ nodes AB*

shows $\exists p . \text{path } AB \ p \ q1 \wedge \text{target } p \ q1 = q2 \wedge \text{length } p \leq \text{card } (\text{nodes } A) * \text{card } (\text{nodes } B)$
 <proof>

lemma *productF-language* :
assumes *productF A B FAIL AB*
and *well-formed A*
and *well-formed B*
and $io \in L \ A \cap L \ B$
shows $io \in L \ AB$
 <proof>

lemma *productF-language-state-intermediate* :
assumes $vs \ @ \ xs \in L \ M2 \cap L \ M1$
and *productF M2 M1 FAIL PM*
and *observable M2*
and *well-formed M2*
and *observable M1*
and *well-formed M1*
obtains $q2 \ q1 \ tr$
where *io-targets PM (initial PM) vs = {(q2,q1)}*
 $\text{path } PM \ (xs \ || \ tr) \ (q2, q1)$
 $\text{length } xs = \text{length } tr$
 <proof>

lemma *sequence-to-failure-reaches-FAIL* :
assumes *sequence-to-failure M1 M2 io*
and *OFSM M1*
and *OFSM M2*
and *productF M2 M1 FAIL PM*
shows $FAIL \in \text{io-targets } PM \ (\text{initial } PM) \ io$
 <proof>

lemma *sequence-to-failure-reaches-FAIL-ob* :
assumes *sequence-to-failure M1 M2 io*
and *OFSM M1*
and *OFSM M2*
and *productF M2 M1 FAIL PM*
shows $\text{io-targets } PM \ (\text{initial } PM) \ io = \{FAIL\}$
 <proof>

lemma *sequence-to-failure-alt-def* :
assumes $\text{io-targets } PM \ (\text{initial } PM) \ io = \{FAIL\}$
and *OFSM M1*
and *OFSM M2*
and *productF M2 M1 FAIL PM*
shows *sequence-to-failure M1 M2 io*
 <proof>

end
theory *ATC*
imports *../FSM/FSM*
begin

3 Adaptive test cases

Adaptive test cases (ATCs) are tree-like structures that label nodes with inputs and edges with outputs such that applying an ATC to some FSM is performed by applying the label of its root node and then applying the ATC connected to the root node by an edge labeled with the observed output of the FSM. The result of such an application is here called an ATC-reaction.

ATCs are here modelled to have edges for every possible output from each non-leaf node. This is not a restriction on the definition of ATCs by Hierons [2] as a missing edge can be expressed by an edge to a leaf.

datatype ('in, 'out) ATC = Leaf | Node 'in 'out \Rightarrow ('in, 'out) ATC

inductive atc-reaction :: ('in, 'out, 'state) FSM \Rightarrow 'state \Rightarrow ('in, 'out) ATC
 \Rightarrow ('in \times 'out) list \Rightarrow bool

where

leaf[*intro!*]: atc-reaction M q1 Leaf [] |

node[*intro!*]: q2 \in succ M (x,y) q1

\Rightarrow atc-reaction M q2 (f y) io

\Rightarrow atc-reaction M q1 (Node x f) ((x,y)#io)

inductive-cases leaf-elim[*elim!*] : atc-reaction M q1 Leaf []

inductive-cases node-elim[*elim!*] : atc-reaction M q1 (Node x f) ((x,y)#io)

3.1 Properties of ATC-reactions

lemma atc-reaction-empty[*simp*] :

assumes atc-reaction M q t []

shows t = Leaf

<proof>

lemma atc-reaction-nonempty-no-leaf :

assumes atc-reaction M q t (Cons a io)

shows t \neq Leaf

<proof>

lemma atc-reaction-nonempty[*elim*] :

assumes atc-reaction M q1 t (Cons (x,y) io)

obtains q2 f

where t = Node x f q2 \in succ M (x,y) q1 atc-reaction M q2 (f y) io

<proof>

lemma atc-reaction-path-ex :

assumes atc-reaction M q1 t io

shows \exists tr . path M (io || tr) q1 \wedge length io = length tr

<proof>

lemma atc-reaction-path[*elim*] :

assumes atc-reaction M q1 t io

obtains tr

where path M (io || tr) q1 length io = length tr

<proof>

3.2 Applicability

An ATC can be applied to an FSM if each node-label is contained in the input alphabet of the FSM.

inductive subtest :: ('in, 'out) ATC \Rightarrow ('in, 'out) ATC \Rightarrow bool **where**

t \in range f \Longrightarrow subtest t (Node x f)

lemma accp-subtest : Wellfounded.accp subtest t

<proof>

definition subtest-rel **where** subtest-rel = {(t, Node x f) | f x t. t \in range f}

lemma subtest-rel-altdef: subtest-rel = {(s, t) | s t. subtest s t}

<proof>

lemma subtest-relI [*intro*]: t \in range f \Longrightarrow (t, Node x f) \in subtest-rel

<proof>

lemma subtest-relI' [*intro*]: t = f y \Longrightarrow (t, Node x f) \in subtest-rel

<proof>

lemma wf-subtest-rel [*simp, intro*]: wf subtest-rel

$\langle proof \rangle$

function *inputs-atc* :: ('a,'b) ATC \Rightarrow 'a set **where**
 inputs-atc Leaf = {} |
 inputs-atc (Node x f) = insert x (\bigcup (image *inputs-atc* (range f)))
 $\langle proof \rangle$
termination $\langle proof \rangle$

fun *applicable* :: ('in, 'out, 'state) FSM \Rightarrow ('in, 'out) ATC \Rightarrow bool **where**
 applicable M t = (*inputs-atc* t \subseteq *inputs* M)

fun *applicable-set* :: ('in, 'out, 'state) FSM \Rightarrow ('in, 'out) ATC set \Rightarrow bool **where**
 applicable-set M Ω = (\forall t \in Ω . *applicable* M t)

lemma *applicable-subtest* :
 assumes *applicable* M (Node x f)
shows *applicable* M (f y)
 $\langle proof \rangle$

3.3 Application function IO

Function IO collects all ATC-reactions of some FSM to some ATC.

fun *IO* :: ('in, 'out, 'state) FSM \Rightarrow 'state \Rightarrow ('in, 'out) ATC \Rightarrow ('in \times 'out) list set **where**
 IO M q t = { tr . *atc-reaction* M q t tr }

fun *IO-set* :: ('in, 'out, 'state) FSM \Rightarrow 'state \Rightarrow ('in, 'out) ATC set \Rightarrow ('in \times 'out) list set
where
 IO-set M q Ω = \bigcup { *IO* M q t | t . t \in Ω }

lemma *IO-language* : *IO* M q t \subseteq *language-state* M q
 $\langle proof \rangle$

lemma *IO-leaf[simp]* : *IO* M q Leaf = {}
 $\langle proof \rangle$

lemma *IO-applicable-nonempty* :
 assumes *applicable* M t
 and *completely-specified* M
 and q1 \in *nodes* M
 shows *IO* M q1 t \neq {}
 $\langle proof \rangle$

lemma *IO-in-language* :
 IO M q t \subseteq *LS* M q
 $\langle proof \rangle$

lemma *IO-set-in-language* :
 IO-set M q Ω \subseteq *LS* M q
 $\langle proof \rangle$

3.4 R-distinguishability

A non-empty ATC r-distinguishes two states of some FSM if there exists no shared ATC-reaction.

fun *r-dist* :: ('in, 'out, 'state) FSM \Rightarrow ('in, 'out) ATC \Rightarrow 'state \Rightarrow 'state \Rightarrow bool **where**
 r-dist M t s1 s2 = (t \neq Leaf \wedge *IO* M s1 t \cap *IO* M s2 t = {})

fun *r-dist-set* :: ('in, 'out, 'state) FSM \Rightarrow ('in, 'out) ATC set \Rightarrow 'state \Rightarrow 'state \Rightarrow bool **where**
 r-dist-set M T s1 s2 = (\exists t \in T . *r-dist* M t s1 s2)

lemma *r-dist-dist* :

assumes *applicable* M t
and *completely-specified* M
and *r-dist* M t $q1$ $q2$
and $q1 \in \text{nodes } M$
shows $q1 \neq q2$
 $\langle \text{proof} \rangle$

lemma *r-dist-set-dist* :
assumes *applicable-set* M Ω
and *completely-specified* M
and *r-dist-set* M Ω $q1$ $q2$
and $q1 \in \text{nodes } M$
shows $q1 \neq q2$
 $\langle \text{proof} \rangle$

lemma *r-dist-set-dist-disjoint* :
assumes *applicable-set* M Ω
and *completely-specified* M
and $\forall t1 \in T1 . \forall t2 \in T2 . \text{r-dist-set } M \ \Omega \ t1 \ t2$
and $T1 \subseteq \text{nodes } M$
shows $T1 \cap T2 = \{\}$
 $\langle \text{proof} \rangle$

3.5 Response sets

The following functions calculate the sets of all ATC-reactions observed by applying some set of ATCs on every state reached in some FSM using a given set of IO-sequences.

fun $B :: ('in, 'out, 'state) \text{FSM} \Rightarrow ('in * 'out) \text{list} \Rightarrow ('in, 'out) \text{ATC set}$
 $\Rightarrow ('in * 'out) \text{list set}$ **where**
 $B \ M \ io \ \Omega = \bigcup (\text{image } (\lambda s . \text{IO-set } M \ s \ \Omega) (\text{io-targets } M \ (\text{initial } M) \ io))$

fun $D :: ('in, 'out, 'state) \text{FSM} \Rightarrow 'in \text{list set} \Rightarrow ('in, 'out) \text{ATC set}$
 $\Rightarrow ('in * 'out) \text{list set set}$ **where**
 $D \ M \ \text{ISeqs } \Omega = \text{image } (\lambda io . B \ M \ io \ \Omega) (\text{LS}_{in} \ M \ (\text{initial } M) \ \text{ISeqs})$

fun *append-io-B* :: $('in, 'out, 'state) \text{FSM} \Rightarrow ('in * 'out) \text{list} \Rightarrow ('in, 'out) \text{ATC set}$
 $\Rightarrow ('in * 'out) \text{list set}$ **where**
 $\text{append-io-B } M \ io \ \Omega = \{ \text{io@res} \mid \text{res} . \text{res} \in B \ M \ io \ \Omega \}$

lemma *B-dist'* :
assumes *df*: $B \ M \ io1 \ \Omega \neq B \ M \ io2 \ \Omega$
shows $(\text{io-targets } M \ (\text{initial } M) \ io1) \neq (\text{io-targets } M \ (\text{initial } M) \ io2)$
 $\langle \text{proof} \rangle$

lemma *B-dist* :
assumes *io-targets* M $(\text{initial } M) \ io1 = \{q1\}$
and *io-targets* M $(\text{initial } M) \ io2 = \{q2\}$
and $B \ M \ io1 \ \Omega \neq B \ M \ io2 \ \Omega$
shows $q1 \neq q2$
 $\langle \text{proof} \rangle$

lemma *D-bound* :
assumes *wf*: *well-formed* M
and *ob*: *observable* M
and *fi*: *finite* ISeqs
shows *finite* $(D \ M \ \text{ISeqs } \Omega)$ $\text{card } (D \ M \ \text{ISeqs } \Omega) \leq \text{card } (\text{nodes } M)$
 $\langle \text{proof} \rangle$

lemma *append-io-B-in-language* :

append-io-B $M \text{ io } \Omega \subseteq L M$
 ⟨proof⟩

lemma *append-io-B-nonempty* :
assumes *applicable-set* $M \Omega$
and *completely-specified* M
and $\text{io} \in \text{language-state } M$ (*initial* M)
and $\Omega \neq \{\}$
shows *append-io-B* $M \text{ io } \Omega \neq \{\}$
 ⟨proof⟩

lemma *append-io-B-prefix-in-language* :
assumes *append-io-B* $M \text{ io } \Omega \neq \{\}$
shows $\text{io} \in L M$
 ⟨proof⟩

3.6 Characterizing sets

A set of ATCs is a characterizing set for some FSM if for every pair of r-distinguishable states it contains an ATC that r-distinguishes them.

fun *characterizing-atc-set* :: (*'in, 'out, 'state*) $FSM \Rightarrow$ (*'in, 'out*) $ATC \text{ set} \Rightarrow \text{bool}$ **where**
characterizing-atc-set $M \Omega = (\text{applicable-set } M \Omega \wedge (\forall s1 \in (\text{nodes } M) . \forall s2 \in (\text{nodes } M) .$
 $(\exists td . r\text{-dist } M td s1 s2) \longrightarrow (\exists tt \in \Omega . r\text{-dist } M tt s1 s2)))$

3.7 Reduction over ATCs

Some state is a an ATC-reduction of another over some set of ATCs if for every contained ATC every ATC-reaction to it of the former state is also an ATC-reaction of the latter state.

fun *atc-reduction* :: (*'in, 'out, 'state*) $FSM \Rightarrow$ *'state* \Rightarrow (*'in, 'out, 'state*) $FSM \Rightarrow$ *'state*
 \Rightarrow (*'in, 'out*) $ATC \text{ set} \Rightarrow \text{bool}$ **where**
atc-reduction $M2 s2 M1 s1 \Omega = (\forall t \in \Omega . IO M2 s2 t \subseteq IO M1 s1 t)$

— r-distinguishability holds for atc-reductions

lemma *atc-rdist-dist[intro]* :
assumes *wf2* : *well-formed* $M2$
and *cs2* : *completely-specified* $M2$
and *ap2* : *applicable-set* $M2 \Omega$
and *el-t1* : $t1 \in \text{nodes } M2$
and *red1* : *atc-reduction* $M2 t1 M1 s1 \Omega$
and *red2* : *atc-reduction* $M2 t2 M1 s2 \Omega$
and *rdist* : *r-dist-set* $M1 \Omega s1 s2$
and $t1 \in \text{nodes } M2$
shows *r-dist-set* $M2 \Omega t1 t2$
 ⟨proof⟩

3.8 Reduction over ATCs applied after input sequences

The following functions check whether some FSM is a reduction of another over a given set of input sequences while furthermore the response sets obtained by applying a set of ATCs after every input sequence to the first FSM are subsets of the analogously constructed response sets of the second FSM.

fun *atc-io-reduction-on* :: (*'in, 'out, 'state1*) $FSM \Rightarrow$ (*'in, 'out, 'state2*) $FSM \Rightarrow$ *'in list*
 \Rightarrow (*'in, 'out*) $ATC \text{ set} \Rightarrow \text{bool}$ **where**
atc-io-reduction-on $M1 M2 \text{iseq } \Omega = (L_{in} M1 \{\text{iseq}\} \subseteq L_{in} M2 \{\text{iseq}\}$
 $\wedge (\forall \text{io} \in L_{in} M1 \{\text{iseq}\} . B M1 \text{io } \Omega \subseteq B M2 \text{io } \Omega))$

fun *atc-io-reduction-on-sets* :: (*'in, 'out, 'state1*) $FSM \Rightarrow$ *'in list set* \Rightarrow (*'in, 'out*) $ATC \text{ set}$
 \Rightarrow (*'in, 'out, 'state2*) $FSM \Rightarrow \text{bool}$ **where**
atc-io-reduction-on-sets $M1 TS \Omega M2 = (\forall \text{iseq} \in TS . \text{atc-io-reduction-on } M1 M2 \text{iseq } \Omega)$

notation

atc-io-reduction-on-sets ($\langle (- \preceq [-] -) \rangle$ [1000,1000,1000,1000])

lemma *io-reduction-from-atc-io-reduction* :
 assumes *atc-io-reduction-on-sets* $M1\ T\ \Omega\ M2$
 and *finite* T
 shows *io-reduction-on* $M1\ T\ M2$
 $\langle proof \rangle$

lemma *atc-io-reduction-on-subset* :
 assumes *atc-io-reduction-on-sets* $M1\ T\ \Omega\ M2$
 and $T' \subseteq T$
shows *atc-io-reduction-on-sets* $M1\ T'\ \Omega\ M2$
 $\langle proof \rangle$

lemma *atc-reaction-reduction*[intro] :
 assumes *ls* : *language-state* $M1\ q1 \subseteq$ *language-state* $M2\ q2$
 and *el1* : $q1 \in$ *nodes* $M1$
 and *el2* : $q2 \in$ *nodes* $M2$
 and *rct* : *atc-reaction* $M1\ q1\ t\ io$
 and *ob2* : *observable* $M2$
 and *ob1* : *observable* $M1$
shows *atc-reaction* $M2\ q2\ t\ io$
 $\langle proof \rangle$

lemma *IO-reduction* :
 assumes *ls* : *language-state* $M1\ q1 \subseteq$ *language-state* $M2\ q2$
 and *el1* : $q1 \in$ *nodes* $M1$
 and *el2* : $q2 \in$ *nodes* $M2$
 and *ob1* : *observable* $M1$
 and *ob2* : *observable* $M2$
shows *IO* $M1\ q1\ t \subseteq$ *IO* $M2\ q2\ t$
 $\langle proof \rangle$

lemma *IO-set-reduction* :
 assumes *ls* : *language-state* $M1\ q1 \subseteq$ *language-state* $M2\ q2$
 and *el1* : $q1 \in$ *nodes* $M1$
 and *el2* : $q2 \in$ *nodes* $M2$
 and *ob1* : *observable* $M1$
 and *ob2* : *observable* $M2$
shows *IO-set* $M1\ q1\ \Omega \subseteq$ *IO-set* $M2\ q2\ \Omega$
 $\langle proof \rangle$

lemma *B-reduction* :
 assumes *red* : $M1 \preceq M2$
 and *ob1* : *observable* $M1$
 and *ob2* : *observable* $M2$
shows *B* $M1\ io\ \Omega \subseteq$ *B* $M2\ io\ \Omega$
 $\langle proof \rangle$

lemma *append-io-B-reduction* :
 assumes *red* : $M1 \preceq M2$
 and *ob1* : *observable* $M1$
 and *ob2* : *observable* $M2$
shows *append-io-B* $M1\ io\ \Omega \subseteq$ *append-io-B* $M2\ io\ \Omega$
 $\langle proof \rangle$

lemma *atc-io-reduction-on-reduction*[intro] :
 assumes *red* : $M1 \preceq M2$
 and *ob1* : *observable* $M1$

```

and    ob2 : observable M2
shows atc-io-reduction-on M1 M2 iseq Ω
⟨proof⟩

```

```

lemma atc-io-reduction-on-sets-reduction[intro] :
  assumes red : M1 ≼ M2
  and    ob1 : observable M1
  and    ob2 : observable M2
shows atc-io-reduction-on-sets M1 TS Ω M2
⟨proof⟩

```

```

lemma atc-io-reduction-on-sets-via-LSin :
  assumes atc-io-reduction-on-sets M1 TS Ω M2
  shows (Lin M1 TS ∪ (⋃io∈Lin M1 TS. B M1 io Ω))
        ⊆ (Lin M2 TS ∪ (⋃io∈Lin M2 TS. B M2 io Ω))
⟨proof⟩

```

```

end
theory ASC-LB
imports ../ATC/ATC ../FSM/FSM-Product
begin

```

4 The lower bound function

This theory defines the lower bound function LB and its properties.

Function LB calculates a lower bound on the number of states of some FSM in order for some sequence to not contain certain repetitions.

4.1 Permutation function Perm

Function Perm calculates all possible reactions of an FSM to a set of inputs sequences such that every set in the calculated set of reactions contains exactly one reaction for each input sequence.

```

fun Perm :: 'in list set ⇒ ('in, 'out, 'state) FSM ⇒ ('in × 'out) list set set where
  Perm V M = {image f V | f . ∀ v ∈ V . f v ∈ language-state-for-input M (initial M) v }

```

```

lemma perm-empty :
  assumes is-det-state-cover M2 V
  and V'' ∈ Perm V M1
shows [] ∈ V''
⟨proof⟩

```

```

lemma perm-elem-finite :
  assumes is-det-state-cover M2 V
  and well-formed M2
  and V'' ∈ Perm V M1
shows finite V''
⟨proof⟩

```

```

lemma perm-inputs :
  assumes V'' ∈ Perm V M
  and vs ∈ V''
shows map fst vs ∈ V
⟨proof⟩

```

```

lemma perm-inputs-diff :
  assumes V'' ∈ Perm V M
  and vs1 ∈ V''
  and vs2 ∈ V''
  and vs1 ≠ vs2
shows map fst vs1 ≠ map fst vs2

```

<proof>

lemma *perm-language* :
 assumes $V'' \in \text{Perm } V M$
 and $vs \in V''$
shows $vs \in L M$
<proof>

4.2 Helper predicates

The following predicates are used to combine often repeated assumption.

abbreviation *asc-fault-domain* $M2 M1 m \equiv (\text{inputs } M2 = \text{inputs } M1 \wedge \text{card } (\text{nodes } M1) \leq m)$

lemma *asc-fault-domain-props*[*elim!*] :
 assumes *asc-fault-domain* $M2 M1 m$
 shows $\text{inputs } M2 = \text{inputs } M1$
 $\text{card } (\text{nodes } M1) \leq m$ *<proof>*

abbreviation
test-tools $M2 M1 \text{ FAIL } PM V \Omega \equiv (
 \text{productF } M2 M1 \text{ FAIL } PM
 \wedge \text{is-det-state-cover } M2 V
 \wedge \text{applicable-set } M2 \Omega
)$

lemma *test-tools-props*[*elim*] :
 assumes *test-tools* $M2 M1 \text{ FAIL } PM V \Omega$
 and *asc-fault-domain* $M2 M1 m$
 shows $\text{productF } M2 M1 \text{ FAIL } PM$
 $\text{is-det-state-cover } M2 V$
 $\text{applicable-set } M2 \Omega$
 $\text{applicable-set } M1 \Omega$
<proof>

lemma *perm-nonempty* :
 assumes $\text{is-det-state-cover } M2 V$
 and *OFSM* $M1$
 and *OFSM* $M2$
 and $\text{inputs } M1 = \text{inputs } M2$
shows $\text{Perm } V M1 \neq \{\}$
<proof>

lemma *perm-elem* :
 assumes $\text{is-det-state-cover } M2 V$
 and *OFSM* $M1$
 and *OFSM* $M2$
 and $\text{inputs } M1 = \text{inputs } M2$
 and $vs \in V$
 and $vs' \in \text{language-state-for-input } M1 (\text{initial } M1) vs$
obtains V''
where $V'' \in \text{Perm } V M1$ $vs' \in V''$
<proof>

4.3 Function R

Function R calculates the set of suffixes of a sequence that reach a given state if applied after a given other sequence.

fun $R :: ('in, 'out, 'state) \text{FSM} \Rightarrow 'state \Rightarrow ('in \times 'out) \text{list}$
 $\Rightarrow ('in \times 'out) \text{list} \Rightarrow ('in \times 'out) \text{list set}$
where

$$R M s vs xs = \{ vs@xs' \mid xs' . xs' \neq [] \\ \wedge \text{prefix } xs' xs \\ \wedge s \in \text{io-targets } M (\text{initial } M) (vs@xs') \}$$

lemma *finite-R* : *finite* (R M s vs xs)
 ⟨proof⟩

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

lemma *card-union-of-distinct* :
assumes $\forall S1 \in SS . \forall S2 \in SS . S1 = S2 \vee f S1 \cap f S2 = \{\}$
and *finite* SS
and $\forall S \in SS . f S \neq \{\}$
shows $\text{card} (\text{image } f SS) = \text{card } SS$
 ⟨proof⟩

lemma *R-count* :
assumes $(vs @ xs) \in L M1 \cap L M2$
and *observable* M1
and *observable* M2
and *well-formed* M1
and *well-formed* M2
and $s \in \text{nodes } M2$
and *productF* M2 M1 FAIL PM
and $\text{io-targets } PM (\text{initial } PM) vs = \{(q2, q1)\}$
and *path* PM (xs || tr) (q2, q1)
and $\text{length } xs = \text{length } tr$
and *distinct* (states (xs || tr) (q2, q1))
shows $\text{card} (\bigcup (\text{image} (\text{io-targets } M1 (\text{initial } M1)) (R M2 s vs xs))) = \text{card} (R M2 s vs xs)$
 — each sequence in the set calculated by R reaches a different state in M1
 ⟨proof⟩

lemma *R-state-component-2* :
assumes $io \in (R M2 s vs xs)$
and *observable* M2
shows $\text{io-targets } M2 (\text{initial } M2) io = \{s\}$
 ⟨proof⟩

lemma *R-union-card-is-suffix-length* :
assumes OFSM M2
and $io@xs \in L M2$
shows $\text{sum} (\lambda q . \text{card} (R M2 q io xs)) (\text{nodes } M2) = \text{length } xs$
 ⟨proof⟩

lemma *R-state-repetition-via-long-sequence* :
assumes OFSM M
and $\text{card} (\text{nodes } M) \leq m$
and $\text{Suc } (m * m) \leq \text{length } xs$
and $vs@xs \in L M$
shows $\exists q \in \text{nodes } M . \text{card} (R M q vs xs) > m$
 ⟨proof⟩

lemma *R-state-repetition-distribution* :
assumes OFSM M
and $\text{Suc} (\text{card} (\text{nodes } M) * m) \leq \text{length } xs$
and $vs@xs \in L M$
shows $\exists q \in \text{nodes } M . \text{card} (R M q vs xs) > m$
 ⟨proof⟩

4.4 Function RP

Function RP extends function MR by adding all elements from a set of IO-sequences that also reach the given state.

```

fun RP :: ('in, 'out, 'state) FSM  $\Rightarrow$  'state  $\Rightarrow$  ('in  $\times$  'out) list
            $\Rightarrow$  ('in  $\times$  'out) list  $\Rightarrow$  ('in  $\times$  'out) list set
            $\Rightarrow$  ('in  $\times$  'out) list set
where
  RP M s vs xs V'' = R M s vs xs
                     $\cup$  {vs'  $\in$  V'' . io-targets M (initial M) vs' = {s}}

```

lemma RP-from-R:

```

assumes is-det-state-cover M2 V
and     V''  $\in$  Perm V M1
shows RP M2 s vs xs V'' = R M2 s vs xs
         $\vee$  ( $\exists$  vs'  $\in$  V'' . vs'  $\notin$  R M2 s vs xs  $\wedge$  RP M2 s vs xs V'' = insert vs' (R M2 s vs xs))
<proof>

```

lemma finite-RP :

```

assumes is-det-state-cover M2 V
and     V''  $\in$  Perm V M1
shows finite (RP M2 s vs xs V'')
<proof>

```

lemma RP-count :

```

assumes (vs @ xs)  $\in$  L M1  $\cap$  L M2
and     observable M1
and     observable M2
and     well-formed M1
and     well-formed M2
and     s  $\in$  nodes M2
and     productF M2 M1 FAIL PM
and     io-targets PM (initial PM) vs = {(q2,q1)}
and     path PM (xs || tr) (q2,q1)
and     length xs = length tr
and     distinct (states (xs || tr) (q2,q1))
and     is-det-state-cover M2 V
and     V''  $\in$  Perm V M1
and      $\forall$  s'  $\in$  set (states (xs || map fst tr) q2) .  $\neg$  ( $\exists$  v  $\in$  V . d-reaches M2 (initial M2) v s')
shows card ( $\bigcup$  (image (io-targets M1 (initial M1)) (RP M2 s vs xs V''))) = card (RP M2 s vs xs V'')
        — each sequence in the set calculated by RP reaches a different state in M1
<proof>

```

lemma RP-state-component-2 :

```

assumes io  $\in$  (RP M2 s vs xs V'')
and     observable M2
shows io-targets M2 (initial M2) io = {s}
<proof>

```

lemma RP-io-targets-split :

```

assumes (vs @ xs)  $\in$  L M1  $\cap$  L M2
and     observable M1
and     observable M2
and     well-formed M1
and     well-formed M2
and     productF M2 M1 FAIL PM
and     is-det-state-cover M2 V
and     V''  $\in$  Perm V M1
and     io  $\in$  RP M2 s vs xs V''
shows io-targets PM (initial PM) io
        = io-targets M2 (initial M2) io  $\times$  io-targets M1 (initial M1) io

```

$\langle \text{proof} \rangle$

lemma *RP-io-targets-finite-M1* :

assumes $(vs @ xs) \in L M1 \cap L M2$

and *observable M1*

and *is-det-state-cover M2 V*

and $V'' \in \text{Perm } V M1$

shows *finite* $(\bigcup (\text{image } (\text{io-targets } M1 (\text{initial } M1)) (RP M2 s vs xs V'')))$

$\langle \text{proof} \rangle$

lemma *RP-io-targets-finite-PM* :

assumes $(vs @ xs) \in L M1 \cap L M2$

and *observable M1*

and *observable M2*

and *well-formed M1*

and *well-formed M2*

and *productF M2 M1 FAIL PM*

and *is-det-state-cover M2 V*

and $V'' \in \text{Perm } V M1$

shows *finite* $(\bigcup (\text{image } (\text{io-targets } PM (\text{initial } PM)) (RP M2 s vs xs V'')))$

$\langle \text{proof} \rangle$

lemma *RP-union-card-is-suffix-length* :

assumes *OFSM M2*

and $\text{io}@xs \in L M2$

and *is-det-state-cover M2 V*

and $V'' \in \text{Perm } V M1$

shows $\bigwedge q . \text{card } (R M2 q \text{ io } xs) \leq \text{card } (RP M2 q \text{ io } xs V'')$
 $\text{sum } (\lambda q . \text{card } (RP M2 q \text{ io } xs V'')) (\text{nodes } M2) \geq \text{length } xs$

$\langle \text{proof} \rangle$

lemma *RP-state-repetition-distribution-productF* :

assumes *OFSM M2*

and *OFSM M1*

and $(\text{card } (\text{nodes } M2) * m) \leq \text{length } xs$

and $\text{card } (\text{nodes } M1) \leq m$

and $vs@xs \in L M2 \cap L M1$

and *is-det-state-cover M2 V*

and $V'' \in \text{Perm } V M1$

shows $\exists q \in \text{nodes } M2 . \text{card } (RP M2 q vs xs V'') > m$

$\langle \text{proof} \rangle$

4.5 Conditions for the result of LB to be a valid lower bound

The following predicates describe the assumptions necessary to show that the value calculated by LB is a lower bound on the number of states of a given FSM.

fun *Prereq* :: $(\text{'in}, \text{'out}, \text{'state1}) \text{ FSM} \Rightarrow (\text{'in}, \text{'out}, \text{'state2}) \text{ FSM} \Rightarrow (\text{'in} \times \text{'out}) \text{ list}$
 $\Rightarrow (\text{'in} \times \text{'out}) \text{ list} \Rightarrow \text{'in list set} \Rightarrow \text{'state1 set} \Rightarrow (\text{'in}, \text{'out}) \text{ ATC set}$
 $\Rightarrow (\text{'in} \times \text{'out}) \text{ list set} \Rightarrow \text{bool}$

where

Prereq M2 M1 vs xs T S Ω V'' = (

(finite T)

$\wedge (vs @ xs) \in L M2 \cap L M1$

$\wedge S \subseteq \text{nodes } M2$

$\wedge (\forall s1 \in S . \forall s2 \in S . s1 \neq s2$

$\longrightarrow (\forall io1 \in RP M2 s1 vs xs V'' .$

$\forall io2 \in RP M2 s2 vs xs V'' .$

$B M1 io1 \Omega \neq B M1 io2 \Omega))$)

```

fun Rep-Pre :: ('in, 'out, 'state1) FSM  $\Rightarrow$  ('in, 'out, 'state2) FSM  $\Rightarrow$  ('in  $\times$  'out) list
               $\Rightarrow$  ('in  $\times$  'out) list  $\Rightarrow$  bool where
  Rep-Pre M2 M1 vs xs = ( $\exists$  xs1 xs2 . prefix xs1 xs2  $\wedge$  prefix xs2 xs  $\wedge$  xs1  $\neq$  xs2
     $\wedge$  ( $\exists$  s2 . io-targets M2 (initial M2) (vs @ xs1) = {s2}
       $\wedge$  io-targets M2 (initial M2) (vs @ xs2) = {s2})
     $\wedge$  ( $\exists$  s1 . io-targets M1 (initial M1) (vs @ xs1) = {s1}
       $\wedge$  io-targets M1 (initial M1) (vs @ xs2) = {s1}))

fun Rep-Cov :: ('in, 'out, 'state1) FSM  $\Rightarrow$  ('in, 'out, 'state2) FSM  $\Rightarrow$  ('in  $\times$  'out) list set
               $\Rightarrow$  ('in  $\times$  'out) list  $\Rightarrow$  ('in  $\times$  'out) list  $\Rightarrow$  bool where
  Rep-Cov M2 M1 V'' vs xs = ( $\exists$  xs' vs' . xs'  $\neq$  []  $\wedge$  prefix xs' xs  $\wedge$  vs'  $\in$  V''
     $\wedge$  ( $\exists$  s2 . io-targets M2 (initial M2) (vs @ xs') = {s2}
       $\wedge$  io-targets M2 (initial M2) (vs') = {s2})
     $\wedge$  ( $\exists$  s1 . io-targets M1 (initial M1) (vs @ xs') = {s1}
       $\wedge$  io-targets M1 (initial M1) (vs') = {s1}))

```

```

lemma distinctness-via-Rep-Pre :
  assumes  $\neg$  Rep-Pre M2 M1 vs xs
  and productF M2 M1 FAIL PM
  and observable M1
  and observable M2
  and io-targets PM (initial PM) vs = {(q2,q1)}
  and path PM (xs || tr) (q2,q1)
  and length xs = length tr
  and (vs @ xs)  $\in$  L M1  $\cap$  L M2
  and well-formed M1
  and well-formed M2
shows distinct (states (xs || tr) (q2, q1))
<proof>

```

```

lemma RP-count-via-Rep-Cov :
  assumes (vs @ xs)  $\in$  L M1  $\cap$  L M2
  and observable M1
  and observable M2
  and well-formed M1
  and well-formed M2
  and s  $\in$  nodes M2
  and productF M2 M1 FAIL PM
  and io-targets PM (initial PM) vs = {(q2,q1)}
  and path PM (xs || tr) (q2,q1)
  and length xs = length tr
  and distinct (states (xs || tr) (q2,q1))
  and is-det-state-cover M2 V
  and V''  $\in$  Perm V M1
  and  $\neg$  Rep-Cov M2 M1 V'' vs xs
shows card ( $\bigcup$  (image (io-targets M1 (initial M1)) (RP M2 s vs xs V''))) = card (RP M2 s vs xs V'')
<proof>

```

```

lemma RP-count-alt-def :
  assumes (vs @ xs)  $\in$  L M1  $\cap$  L M2
  and observable M1
  and observable M2
  and well-formed M1
  and well-formed M2
  and s  $\in$  nodes M2
  and productF M2 M1 FAIL PM
  and io-targets PM (initial PM) vs = {(q2,q1)}

```

and *path* $PM (xs \parallel tr) (q2, q1)$
and $length\ xs = length\ tr$
and $\neg Rep\text{-}Pre\ M2\ M1\ vs\ xs$
and *is-det-state-cover* $M2\ V$
and $V'' \in Perm\ V\ M1$
and $\neg Rep\text{-}Cov\ M2\ M1\ V''\ vs\ xs$
shows $card\ (\bigcup (image\ (io\text{-}targets\ M1\ (initial\ M1))\ (RP\ M2\ s\ vs\ xs\ V'')) = card\ (RP\ M2\ s\ vs\ xs\ V'')$
 $\langle proof \rangle$

4.6 Function LB

LB adds together the number of elements in sets calculated via RP for a given set of states and the number of ATC-reaction known to exist but not produced by a state reached by any of the above elements.

fun $LB :: ('in, 'out, 'state1)\ FSM \Rightarrow ('in, 'out, 'state2)\ FSM$
 $\Rightarrow ('in \times 'out)\ list \Rightarrow ('in \times 'out)\ list \Rightarrow 'in\ list\ set$
 $\Rightarrow 'state1\ set \Rightarrow ('in, 'out)\ ATC\ set$
 $\Rightarrow ('in \times 'out)\ list\ set \Rightarrow nat$
where
 $LB\ M2\ M1\ vs\ xs\ T\ S\ \Omega\ V'' =$
 $(sum\ (\lambda\ s.\ card\ (RP\ M2\ s\ vs\ xs\ V''))\ S)$
 $+ card\ ((D\ M1\ T\ \Omega) -$
 $\{B\ M1\ xs'\ \Omega \mid xs'\ s' . s' \in S \wedge xs' \in RP\ M2\ s'\ vs\ xs\ V''\})$

lemma *LB-count-helper-RP-disjoint-and-cards* :

assumes $(vs @ xs) \in L\ M1 \cap L\ M2$
and *observable* $M1$
and *observable* $M2$
and *well-formed* $M1$
and *well-formed* $M2$
and *productF* $M2\ M1\ FAIL\ PM$
and *is-det-state-cover* $M2\ V$
and $V'' \in Perm\ V\ M1$
and $s1 \neq s2$
shows $\bigcup (image\ (io\text{-}targets\ PM\ (initial\ PM))\ (RP\ M2\ s1\ vs\ xs\ V''))$
 $\cap \bigcup (image\ (io\text{-}targets\ PM\ (initial\ PM))\ (RP\ M2\ s2\ vs\ xs\ V'')) = \{\}$
 $card\ (\bigcup (image\ (io\text{-}targets\ PM\ (initial\ PM))\ (RP\ M2\ s1\ vs\ xs\ V'')))$
 $= card\ (\bigcup (image\ (io\text{-}targets\ M1\ (initial\ M1))\ (RP\ M2\ s1\ vs\ xs\ V'')))$
 $card\ (\bigcup (image\ (io\text{-}targets\ PM\ (initial\ PM))\ (RP\ M2\ s2\ vs\ xs\ V'')))$
 $= card\ (\bigcup (image\ (io\text{-}targets\ M1\ (initial\ M1))\ (RP\ M2\ s2\ vs\ xs\ V'')))$
 $\langle proof \rangle$

lemma *LB-count-helper-RP-disjoint-card-M1* :

assumes $(vs @ xs) \in L\ M1 \cap L\ M2$
and *observable* $M1$
and *observable* $M2$
and *well-formed* $M1$
and *well-formed* $M2$
and *productF* $M2\ M1\ FAIL\ PM$
and *is-det-state-cover* $M2\ V$
and $V'' \in Perm\ V\ M1$
and $s1 \neq s2$
shows $card\ (\bigcup (image\ (io\text{-}targets\ PM\ (initial\ PM))\ (RP\ M2\ s1\ vs\ xs\ V''))$
 $\cup \bigcup (image\ (io\text{-}targets\ PM\ (initial\ PM))\ (RP\ M2\ s2\ vs\ xs\ V'')))$
 $= card\ (\bigcup (image\ (io\text{-}targets\ M1\ (initial\ M1))\ (RP\ M2\ s1\ vs\ xs\ V'')))$
 $+ card\ (\bigcup (image\ (io\text{-}targets\ M1\ (initial\ M1))\ (RP\ M2\ s2\ vs\ xs\ V'')))$
 $\langle proof \rangle$

lemma *LB-count-helper-RP-disjoint-M1-pair* :

assumes $(vs @ xs) \in L\ M1 \cap L\ M2$
and *observable* $M1$
and *observable* $M2$
and *well-formed* $M1$

and *well-formed* $M2$
and *productF* $M2 M1 FAIL PM$
and *io-targets* $PM (initial PM) vs = \{(q2, q1)\}$
and *path* $PM (xs \parallel tr) (q2, q1)$
and *length* $xs = length tr$
and $\neg Rep-Pre M2 M1 vs xs$
and *is-det-state-cover* $M2 V$
and $V'' \in Perm V M1$
and $\neg Rep-Cov M2 M1 V'' vs xs$
and *Prereq* $M2 M1 vs xs T S \Omega V''$
and $s1 \neq s2$
and $s1 \in S$
and $s2 \in S$
and *applicable-set* $M1 \Omega$
and *completely-specified* $M1$
shows $card (RP M2 s1 vs xs V'') + card (RP M2 s2 vs xs V'')$
 $= card (\bigcup (image (io-targets M1 (initial M1)) (RP M2 s1 vs xs V'')))$
 $+ card (\bigcup (image (io-targets M1 (initial M1)) (RP M2 s2 vs xs V'')))$
 $\bigcup (image (io-targets M1 (initial M1)) (RP M2 s1 vs xs V''))$
 $\cap \bigcup (image (io-targets M1 (initial M1)) (RP M2 s2 vs xs V''))$
 $= \{\}$
 $\langle proof \rangle$

lemma *LB-count-helper-RP-card-union* :
assumes *observable* $M2$
and $s1 \neq s2$
shows $RP M2 s1 vs xs V'' \cap RP M2 s2 vs xs V'' = \{\}$
 $\langle proof \rangle$

lemma *LB-count-helper-RP-inj* :
obtains f
where $\forall q \in (\bigcup (image (\lambda s . \bigcup (image (io-targets M1 (initial M1)) (RP M2 s vs xs V'')) S)) .$
 $f q \in nodes M1$
 $inj-on f (\bigcup (image (\lambda s . \bigcup (image (io-targets M1 (initial M1)) (RP M2 s vs xs V'')) S))$
 $\langle proof \rangle$

abbreviation (*input*) *UNION* :: $'a set \Rightarrow ('a \Rightarrow 'b set) \Rightarrow 'b set$
where $UNION A f \equiv \bigcup (f \text{ ` } A)$

lemma *LB-count-helper-RP-card-union-sum* :
assumes $(vs @ xs) \in L M2 \cap L M1$
and *OFSM* $M1$
and *OFSM* $M2$
and *asc-fault-domain* $M2 M1 m$
and *test-tools* $M2 M1 FAIL PM V \Omega$
and $V'' \in Perm V M1$
and *Prereq* $M2 M1 vs xs T S \Omega V''$
and $\neg Rep-Pre M2 M1 vs xs$
and $\neg Rep-Cov M2 M1 V'' vs xs$
shows $sum (\lambda s . card (RP M2 s vs xs V'')) S$
 $= sum (\lambda s . card (\bigcup (image (io-targets M1 (initial M1)) (RP M2 s vs xs V'')) S$
 $\langle proof \rangle$

lemma *finite-insert-card* :
assumes *finite* $(\bigcup SS)$
and *finite* S
and $S \cap (\bigcup SS) = \{\}$

shows $\text{card} (\bigcup (\text{insert } S \ SS)) = \text{card} (\bigcup \ SS) + \text{card } S$
 ⟨proof⟩

lemma *LB-count-helper-RP-disjoint-M1-union* :

assumes $(vs \ @ \ xs) \in L \ M2 \cap L \ M1$
and *OFSM* $M1$
and *OFSM* $M2$
and *asc-fault-domain* $M2 \ M1 \ m$
and *test-tools* $M2 \ M1 \ FAIL \ PM \ V \ \Omega$
and $V'' \in \text{Perm } V \ M1$
and *Prereq* $M2 \ M1 \ vs \ xs \ T \ S \ \Omega \ V''$
and $\neg \text{Rep-Pre } M2 \ M1 \ vs \ xs$
and $\neg \text{Rep-Cov } M2 \ M1 \ V'' \ vs \ xs$
shows $\text{sum } (\lambda \ s \ . \ \text{card} (\text{RP } M2 \ s \ vs \ xs \ V'')) \ S$
 $= \text{card} (\bigcup (\text{image } (\lambda \ s \ . \ \bigcup (\text{image } (\text{io-targets } M1 \ (\text{initial } M1)) (\text{RP } M2 \ s \ vs \ xs \ V'')) \ S))$
 ⟨proof⟩

lemma *LB-count-helper-LB1* :

assumes $(vs \ @ \ xs) \in L \ M2 \cap L \ M1$
and *OFSM* $M1$
and *OFSM* $M2$
and *asc-fault-domain* $M2 \ M1 \ m$
and *test-tools* $M2 \ M1 \ FAIL \ PM \ V \ \Omega$
and $V'' \in \text{Perm } V \ M1$
and *Prereq* $M2 \ M1 \ vs \ xs \ T \ S \ \Omega \ V''$
and $\neg \text{Rep-Pre } M2 \ M1 \ vs \ xs$
and $\neg \text{Rep-Cov } M2 \ M1 \ V'' \ vs \ xs$
shows $(\text{sum } (\lambda \ s \ . \ \text{card} (\text{RP } M2 \ s \ vs \ xs \ V'')) \ S) \leq \text{card} (\text{nodes } M1)$
 ⟨proof⟩

lemma *LB-count-helper-D-states* :

assumes *observable* M
and $RS \in (D \ M \ T \ \Omega)$
obtains q
where $q \in \text{nodes } M \wedge RS = \text{IO-set } M \ q \ \Omega$
 ⟨proof⟩

lemma *LB-count-helper-LB2* :

assumes *observable* $M1$
and $\text{IO-set } M1 \ q \ \Omega \in (D \ M1 \ T \ \Omega) - \{B \ M1 \ xs' \ \Omega \mid xs' \ s' \ . \ s' \in S \wedge xs' \in \text{RP } M2 \ s' \ vs \ xs \ V''\}$
shows $q \notin (\bigcup (\text{image } (\lambda \ s \ . \ \bigcup (\text{image } (\text{io-targets } M1 \ (\text{initial } M1)) (\text{RP } M2 \ s \ vs \ xs \ V'')) \ S))$
 ⟨proof⟩

4.7 Validity of the result of LB constituting a lower bound

lemma *LB-count* :

assumes $(vs \ @ \ xs) \in L \ M1$
and *OFSM* $M1$
and *OFSM* $M2$
and *asc-fault-domain* $M2 \ M1 \ m$
and *test-tools* $M2 \ M1 \ FAIL \ PM \ V \ \Omega$
and $V'' \in \text{Perm } V \ M1$
and *Prereq* $M2 \ M1 \ vs \ xs \ T \ S \ \Omega \ V''$
and $\neg \text{Rep-Pre } M2 \ M1 \ vs \ xs$
and $\neg \text{Rep-Cov } M2 \ M1 \ V'' \ vs \ xs$
shows $LB \ M2 \ M1 \ vs \ xs \ T \ S \ \Omega \ V'' \leq |M1|$
 ⟨proof⟩

```

lemma contradiction-via-LB :
assumes  $(vs @ xs) \in L M1$ 
and OFSM M1
and OFSM M2
and asc-fault-domain M2 M1 m
and test-tools M2 M1 FAIL PM V  $\Omega$ 
and  $V'' \in Perm V M1$ 
and Prereq M2 M1 vs xs T S  $\Omega$  V''
and  $\neg Rep-Pre M2 M1 vs xs$ 
and  $\neg Rep-Cov M2 M1 V'' vs xs$ 
and  $LB M2 M1 vs xs T S \Omega V'' > m$ 
shows False
<proof>

```

```

end
theory ASC-Suite
imports ASC-LB
begin

```

5 Test suite generated by the Adaptive State Counting Algorithm

5.1 Maximum length contained prefix

```

fun mcp :: 'a list  $\Rightarrow$  'a list set  $\Rightarrow$  'a list  $\Rightarrow$  bool where
  mcp z W p = (prefix p z  $\wedge$  p  $\in$  W  $\wedge$ 
     $(\forall p' . (\text{prefix } p' z \wedge p' \in W) \longrightarrow \text{length } p' \leq \text{length } p)$ )

```

```

lemma mcp-ex :
assumes  $\square \in W$ 
and finite W
obtains p
where mcp z W p
<proof>

```

```

lemma mcp-unique :
assumes mcp z W p
and mcp z W p'
shows p = p'
<proof>

```

```

fun mcp' :: 'a list  $\Rightarrow$  'a list set  $\Rightarrow$  'a list where
  mcp' z W = (THE p . mcp z W p)

```

```

lemma mcp'-intro :
assumes mcp z W p
shows mcp' z W = p
<proof>

```

```

lemma mcp-prefix-of-suffix :
assumes mcp (vs@xs) V vs
and prefix xs' xs
shows mcp (vs@xs') V vs
<proof>

```

```

lemma minimal-sequence-to-failure-extending-mcp :
assumes OFSM M1
and OFSM M2
and is-det-state-cover M2 V
and minimal-sequence-to-failure-extending V M1 M2 vs xs
shows mcp (map fst (vs@xs)) V (map fst vs)
<proof>

```

5.2 Function N

Function N narrows the sets of reaction to the deterministic state cover considered by the adaptive state counting algorithm to contain only relevant sequences. It is the main refinement of the original formulation of the algorithm as given in [2]. An example for the necessity for this refinement is given in [3].

```
fun N :: ('in × 'out) list ⇒ ('in, 'out, 'state) FSM ⇒ 'in list set ⇒ ('in × 'out) list set set
where
  N io M V = { V'' ∈ Perm V M . (map fst (mcp' io V'')) = (mcp' (map fst io) V) }
```

lemma *N-nonempty* :

```
assumes is-det-state-cover M2 V
and OFSM M1
and OFSM M2
and asc-fault-domain M2 M1 m
and io ∈ L M1
shows N io M1 V ≠ {}
⟨proof⟩
```

lemma *N-mcp-prefix* :

```
assumes map fst vs = mcp' (map fst (vs@xs)) V
and V'' ∈ N (vs@xs) M1 V
and is-det-state-cover M2 V
and well-formed M2
and finite V
shows vs ∈ V'' vs = mcp' (vs@xs) V''
⟨proof⟩
```

5.3 Functions TS, C, RM

Function TTS defines the calculation of the test suite used by the adaptive state counting algorithm in an iterative way. It is defined using the three functions TS, C and RM where TS represents the test suite calculated up to some iteration, C contains the sequences considered for extension in some iteration, and RM contains the sequences of the corresponding C result that are not to be extended, which we also call removed sequences.

abbreviation *append-set* :: 'a list set ⇒ 'a set ⇒ 'a list set **where**
append-set T X ≡ {xs @ [x] | xs x . xs ∈ T ∧ x ∈ X}

abbreviation *append-sets* :: 'a list set ⇒ 'a list set ⇒ 'a list set **where**
append-sets T X ≡ {xs @ xs' | xs xs' . xs ∈ T ∧ xs' ∈ X}

```
fun TS :: ('in, 'out, 'state1) FSM ⇒ ('in, 'out, 'state2) FSM
  ⇒ ('in, 'out) ATC set ⇒ 'in list set ⇒ nat ⇒ nat
  ⇒ 'in list set
```

```
and C :: ('in, 'out, 'state1) FSM ⇒ ('in, 'out, 'state2) FSM
  ⇒ ('in, 'out) ATC set ⇒ 'in list set ⇒ nat ⇒ nat
  ⇒ 'in list set
```

```
and RM :: ('in, 'out, 'state1) FSM ⇒ ('in, 'out, 'state2) FSM
  ⇒ ('in, 'out) ATC set ⇒ 'in list set ⇒ nat ⇒ nat
  ⇒ 'in list set
```

where

```
RM M2 M1 Ω V m 0 = {} |
TS M2 M1 Ω V m 0 = {} |
TS M2 M1 Ω V m (Suc 0) = V |
C M2 M1 Ω V m 0 = {} |
C M2 M1 Ω V m (Suc 0) = V |
RM M2 M1 Ω V m (Suc n) =
  {xs' ∈ C M2 M1 Ω V m (Suc n) .
    (¬ (Lin M1 {xs'} ⊆ Lin M2 {xs'}))
  } ∨ (∀ io ∈ Lin M1 {xs'} .
  ∃ V'' ∈ N io M1 V .
  ∃ S1 .
  ∃ vs xs .
  io = (vs@xs)
  ∧ mcp (vs@xs) V'' vs
```

$$\begin{aligned}
& \wedge S1 \subseteq \text{nodes } M2 \\
& \wedge (\forall s1 \in S1 . \forall s2 \in S1 . \\
& \quad s1 \neq s2 \longrightarrow \\
& \quad (\forall io1 \in RP\ M2\ s1\ vs\ xs\ V'' . \\
& \quad \quad \forall io2 \in RP\ M2\ s2\ vs\ xs\ V'' . \\
& \quad \quad B\ M1\ io1\ \Omega \neq B\ M1\ io2\ \Omega)) \\
& \wedge m < LB\ M2\ M1\ vs\ xs\ (TS\ M2\ M1\ \Omega\ V\ m\ n \cup V)\ S1\ \Omega\ V'' \} | \\
C\ M2\ M1\ \Omega\ V\ m\ (Suc\ n) = & \\
(\text{append-set } ((C\ M2\ M1\ \Omega\ V\ m\ n) - (RM\ M2\ M1\ \Omega\ V\ m\ n)) (\text{inputs } M2)) & \\
- (TS\ M2\ M1\ \Omega\ V\ m\ n) | & \\
TS\ M2\ M1\ \Omega\ V\ m\ (Suc\ n) = & \\
(TS\ M2\ M1\ \Omega\ V\ m\ n) \cup (C\ M2\ M1\ \Omega\ V\ m\ (Suc\ n)) &
\end{aligned}$$

abbreviation *lists-of-length* :: 'a set \Rightarrow nat \Rightarrow 'a list set **where**
lists-of-length X n \equiv {xs . length xs = n \wedge set xs \subseteq X}

lemma *append-lists-of-length-alt-def* :
append-sets T (*lists-of-length* X (Suc n)) = *append-set* (*append-sets* T (*lists-of-length* X n)) X
<proof>

5.4 Basic properties of the test suite calculation functions

lemma *C-step* :
assumes n > 0
shows C M2 M1 Ω V m (Suc n) \subseteq (*append-set* (C M2 M1 Ω V m n) (*inputs* M2)) - C M2 M1 Ω V m n
<proof>

lemma *C-extension* :
C M2 M1 Ω V m (Suc n) \subseteq *append-sets* V (*lists-of-length* (*inputs* M2) n)
<proof>

lemma *TS-union* :
shows TS M2 M1 Ω V m i = ($\bigcup j \in (\text{set } [0..<Suc\ i]) . C\ M2\ M1\ \Omega\ V\ m\ j$)
<proof>

lemma *C-disj-le-gz* :
assumes i \leq j
and 0 < i
shows C M2 M1 Ω V m i \cap C M2 M1 Ω V m (Suc j) = {}
<proof>

lemma *C-disj-lt* :
assumes i < j
shows C M2 M1 Ω V m i \cap C M2 M1 Ω V m j = {}
<proof>

lemma *C-disj* :
assumes i \neq j
shows C M2 M1 Ω V m i \cap C M2 M1 Ω V m j = {}
<proof>

lemma *RM-subset* : RM M2 M1 Ω V m i \subseteq C M2 M1 Ω V m i
<proof>

lemma *RM-disj* :
assumes i \leq j

and $0 < i$
shows $RM\ M2\ M1\ \Omega\ V\ m\ i \cap RM\ M2\ M1\ \Omega\ V\ m\ (Suc\ j) = \{\}$
 $\langle proof \rangle$

lemma *T-extension* :
assumes $n > 0$
shows $TS\ M2\ M1\ \Omega\ V\ m\ (Suc\ n) - TS\ M2\ M1\ \Omega\ V\ m\ n$
 $\subseteq (append\text{-}set\ (TS\ M2\ M1\ \Omega\ V\ m\ n)\ (inputs\ M2)) - TS\ M2\ M1\ \Omega\ V\ m\ n$
 $\langle proof \rangle$

lemma *append-set-prefix* :
assumes $xs \in append\text{-}set\ T\ X$
shows $butlast\ xs \in T$
 $\langle proof \rangle$

lemma *C-subset* : $C\ M2\ M1\ \Omega\ V\ m\ i \subseteq TS\ M2\ M1\ \Omega\ V\ m\ i$
 $\langle proof \rangle$

lemma *TS-subset* :
assumes $i \leq j$
shows $TS\ M2\ M1\ \Omega\ V\ m\ i \subseteq TS\ M2\ M1\ \Omega\ V\ m\ j$
 $\langle proof \rangle$

lemma *C-immediate-prefix-containment* :
assumes $vs@xs \in C\ M2\ M1\ \Omega\ V\ m\ (Suc\ (Suc\ i))$
and $xs \neq []$
shows $vs@(butlast\ xs) \in C\ M2\ M1\ \Omega\ V\ m\ (Suc\ i) - RM\ M2\ M1\ \Omega\ V\ m\ (Suc\ i)$
 $\langle proof \rangle$

lemma *TS-immediate-prefix-containment* :
assumes $vs@xs \in TS\ M2\ M1\ \Omega\ V\ m\ i$
and $mcp\ (vs@xs)\ V\ vs$
and $0 < i$
shows $vs@(butlast\ xs) \in TS\ M2\ M1\ \Omega\ V\ m\ i$
 $\langle proof \rangle$

lemma *TS-prefix-containment* :
assumes $vs@xs \in TS\ M2\ M1\ \Omega\ V\ m\ i$
and $mcp\ (vs@xs)\ V\ vs$
and $prefix\ xs'\ xs$
shows $vs@xs' \in TS\ M2\ M1\ \Omega\ V\ m\ i$

— Proof sketch: Perform induction on length difference, as from each prefix it is possible to deduce the desired property for the prefix one element smaller than it via above results
 $\langle proof \rangle$

lemma *C-index* :
assumes $vs\ @\ xs \in C\ M2\ M1\ \Omega\ V\ m\ i$
and $mcp\ (vs@xs)\ V\ vs$
shows $Suc\ (length\ xs) = i$
 $\langle proof \rangle$

lemma *TS-index* :
assumes $vs @ xs \in TS\ M2\ M1\ \Omega\ V\ m\ i$
and $mcp\ (vs@xs)\ V\ vs$
shows $Suc\ (length\ xs) \leq i \implies vs@xs \in C\ M2\ M1\ \Omega\ V\ m\ (Suc\ (length\ xs))$
 $\langle proof \rangle$

lemma *C-extension-options* :
assumes $vs @ xs \in C\ M2\ M1\ \Omega\ V\ m\ i$
and $mcp\ (vs @ xs @ [x])\ V\ vs$
and $x \in inputs\ M2$
and $0 < i$
shows $vs@xs@[x] \in C\ M2\ M1\ \Omega\ V\ m\ (Suc\ i) \vee vs@xs \in RM\ M2\ M1\ \Omega\ V\ m\ i$
 $\langle proof \rangle$

lemma *TS-non-containment-causes* :
assumes $vs@xs \notin TS\ M2\ M1\ \Omega\ V\ m\ i$
and $mcp\ (vs@xs)\ V\ vs$
and $set\ xs \subseteq inputs\ M2$
and $0 < i$
shows $(\exists\ xr\ j . xr \neq xs \wedge prefix\ xr\ xs \wedge j \leq i \wedge vs@xr \in RM\ M2\ M1\ \Omega\ V\ m\ j)$
 $\vee (\exists\ xc . xc \neq xs \wedge prefix\ xc\ xs \wedge vs@xc \in (C\ M2\ M1\ \Omega\ V\ m\ i) - (RM\ M2\ M1\ \Omega\ V\ m\ i))$
(is $?PrefPreviouslyRemoved \vee ?PrefJustContained$
 $\neg ((\exists\ xr\ j . xr \neq xs \wedge prefix\ xr\ xs \wedge j \leq i \wedge vs@xr \in RM\ M2\ M1\ \Omega\ V\ m\ j)$
 $\wedge (\exists\ xc . xc \neq xs \wedge prefix\ xc\ xs \wedge vs@xc \in (C\ M2\ M1\ \Omega\ V\ m\ i) - (RM\ M2\ M1\ \Omega\ V\ m\ i)))$
— If a sequence is not contained in TS up to (incl.) iteration i, then either a prefix of it has been removed or a prefix of it is contained in the C set for iteration i
 $\langle proof \rangle$

lemma *TS-non-containment-causes-rev* :
assumes $mcp\ (vs@xs)\ V\ vs$
and $(\exists\ xr\ j . xr \neq xs \wedge prefix\ xr\ xs \wedge j \leq i \wedge vs@xr \in RM\ M2\ M1\ \Omega\ V\ m\ j)$
 $\vee (\exists\ xc . xc \neq xs \wedge prefix\ xc\ xs \wedge vs@xc \in (C\ M2\ M1\ \Omega\ V\ m\ i) - (RM\ M2\ M1\ \Omega\ V\ m\ i))$
(is $?PrefPreviouslyRemoved \vee ?PrefJustContained$
shows $vs@xs \notin TS\ M2\ M1\ \Omega\ V\ m\ i$
 $\langle proof \rangle$

lemma *TS-finite* :
assumes $finite\ V$
and $finite\ (inputs\ M2)$
shows $finite\ (TS\ M2\ M1\ \Omega\ V\ m\ n)$
 $\langle proof \rangle$

lemma *C-finite* :
assumes $finite\ V$
and $finite\ (inputs\ M2)$
shows $finite\ (C\ M2\ M1\ \Omega\ V\ m\ n)$
 $\langle proof \rangle$

5.5 Final iteration

The result of calculating TS for some iteration is final if the result does not change for the next iteration. Such a final iteration exists and is at most equal to the number of states of FSM M2 multiplied by an upper bound on the number of states of FSM M1.

Furthermore, for any sequence not contained in the final iteration of the test suite, a prefix of this sequence must be contained in the latter.

abbreviation *final-iteration* $M2\ M1\ \Omega\ V\ m\ i \equiv TS\ M2\ M1\ \Omega\ V\ m\ i = TS\ M2\ M1\ \Omega\ V\ m\ (Suc\ i)$

lemma *final-iteration-ex* :
assumes *OFSM* $M1$
and *OFSM* $M2$
and *asc-fault-domain* $M2\ M1\ m$
and *test-tools* $M2\ M1\ FAIL\ PM\ V\ \Omega$
shows *final-iteration* $M2\ M1\ \Omega\ V\ m\ (Suc\ (|M2| * m))$
 $\langle proof \rangle$

lemma *TS-non-containment-causes-final* :
assumes $vs@xs \notin TS\ M2\ M1\ \Omega\ V\ m\ i$
and *mcp* $(vs@xs)\ V\ vs$
and *set* $xs \subseteq inputs\ M2$
and *final-iteration* $M2\ M1\ \Omega\ V\ m\ i$
and *OFSM* $M2$
shows $(\exists\ xr\ j . xr \neq xs$
 $\wedge prefix\ xr\ xs$
 $\wedge j \leq i$
 $\wedge vs@xr \in RM\ M2\ M1\ \Omega\ V\ m\ j)$
 $\langle proof \rangle$

lemma *TS-non-containment-causes-final-suc* :
assumes $vs@xs \notin TS\ M2\ M1\ \Omega\ V\ m\ i$
and *mcp* $(vs@xs)\ V\ vs$
and *set* $xs \subseteq inputs\ M2$
and *final-iteration* $M2\ M1\ \Omega\ V\ m\ i$
and *OFSM* $M2$
obtains $xr\ j$
where $xr \neq xs\ prefix\ xr\ xs\ Suc\ j \leq i\ vs@xr \in RM\ M2\ M1\ \Omega\ V\ m\ (Suc\ j)$
 $\langle proof \rangle$

end
theory *ASC-Sufficiency*
imports *ASC-Suite*
begin

6 Sufficiency of the test suite to test for reduction

This section provides a proof that the test suite generated by the adaptive state counting algorithm is sufficient to test for reduction.

6.1 Properties of minimal sequences to failures extending the deterministic state cover

The following two lemmata show that minimal sequences to failures extending the deterministic state cover do not with their extending suffix visit any state twice or visit a state also reached by a sequence in the chosen permutation of reactions to the deterministic state cover.

lemma *minimal-sequence-to-failure-extending-implies-Rep-Pre* :
assumes *minimal-sequence-to-failure-extending* $V\ M1\ M2\ vs\ xs$
and *OFSM* $M1$
and *OFSM* $M2$
and *test-tools* $M2\ M1\ FAIL\ PM\ V\ \Omega$
and $V'' \in N\ (vs@xs')\ M1\ V$
and *prefix* $xs'\ xs$
shows $\neg Rep-Pre\ M2\ M1\ vs\ xs'$

<proof>

lemma *minimal-sequence-to-failure-extending-implies-Rep-Cov* :
 assumes *minimal-sequence-to-failure-extending* V $M1$ $M2$ vs xs
 and *OFSM* $M1$
 and *OFSM* $M2$
 and *test-tools* $M2$ $M1$ *FAIL* PM V Ω
 and $V'' \in N$ (*vs@xsR*) $M1$ V
 and *prefix* xsR xs
shows \neg *Rep-Cov* $M2$ $M1$ V'' vs xsR
<proof>

lemma *mstfe-no-repetition* :
 assumes *minimal-sequence-to-failure-extending* V $M1$ $M2$ vs xs
 and *OFSM* $M1$
 and *OFSM* $M2$
 and *test-tools* $M2$ $M1$ *FAIL* PM V Ω
 and $V'' \in N$ (*vs@xs'*) $M1$ V
 and *prefix* xs' xs
shows \neg *Rep-Pre* $M2$ $M1$ vs xs'
 and \neg *Rep-Cov* $M2$ $M1$ V'' vs xs'
<proof>

6.2 Sufficiency of the test suite to test for reduction

The following lemma proves that set of input sequences generated in the final iteration of the TS function constitutes a test suite sufficient to test for reduction the FSMs it has been generated for.

This proof is performed by contradiction: If the test suite is not sufficient, then some minimal sequence to a failure extending the deterministic state cover must exist. Due to the test suite being assumed insufficient, this sequence cannot be contained in it and hence a prefix of it must have been contained in one of the sets calculated by the R function. This is only possible if the prefix is not a minimal sequence to a failure extending the deterministic state cover or if the test suite observes a failure, both of which violates the assumptions.

lemma *asc-sufficiency* :
 assumes *OFSM* $M1$
 and *OFSM* $M2$
 and *asc-fault-domain* $M2$ $M1$ m
 and *test-tools* $M2$ $M1$ *FAIL* PM V Ω
 and *final-iteration* $M2$ $M1$ Ω V m i
shows $M1 \preceq [(TS\ M2\ M1\ \Omega\ V\ m\ i) . \Omega] M2 \longrightarrow M1 \preceq M2$
<proof>

6.3 Main result

The following lemmata add to the previous result to show that some FSM $M1$ is a reduction of FSM $M2$ if and only if it is a reduction on the test suite generated by the adaptive state counting algorithm for these FSMs.

lemma *asc-soundness* :
 assumes *OFSM* $M1$
 and *OFSM* $M2$
shows $M1 \preceq M2 \longrightarrow$ *atc-io-reduction-on-sets* $M1$ T Ω $M2$
<proof>

lemma *asc-main-theorem* :
 assumes *OFSM* $M1$
 and *OFSM* $M2$
 and *asc-fault-domain* $M2$ $M1$ m
 and *test-tools* $M2$ $M1$ *FAIL* PM V Ω

and *final-iteration* $M2\ M1\ \Omega\ V\ m\ i$
shows $M1 \preceq M2 \iff \text{atc-io-reduction-on-sets } M1\ (TS\ M2\ M1\ \Omega\ V\ m\ i)\ \Omega\ M2$
 ⟨*proof*⟩

end
theory *ASC-Hoare*
imports *ASC-Sufficiency HOL-Hoare.Hoare-Logic*
begin

7 Correctness of the Adaptive State Counting Algorithm in Hoare-Logic

In this section we give an example implementation of the adaptive state counting algorithm in a simple WHILE-language and prove that this implementation produces a certain output if and only if input FSM $M1$ is a reduction of input FSM $M2$.

lemma *atc-io-reduction-on-sets-from-obs* :
assumes $L_{in}\ M1\ T \subseteq L_{in}\ M2\ T$
and $(\bigcup_{io \in L_{in}} M1\ T. \{io\} \times B\ M1\ io\ \Omega) \subseteq (\bigcup_{io \in L_{in}} M2\ T. \{io\} \times B\ M2\ io\ \Omega)$
shows *atc-io-reduction-on-sets* $M1\ T\ \Omega\ M2$
 ⟨*proof*⟩

lemma *atc-io-reduction-on-sets-to-obs* :
assumes *atc-io-reduction-on-sets* $M1\ T\ \Omega\ M2$
shows $L_{in}\ M1\ T \subseteq L_{in}\ M2\ T$
and $(\bigcup_{io \in L_{in}} M1\ T. \{io\} \times B\ M1\ io\ \Omega) \subseteq (\bigcup_{io \in L_{in}} M2\ T. \{io\} \times B\ M2\ io\ \Omega)$
 ⟨*proof*⟩

lemma *atc-io-reduction-on-sets-alt-def* :
shows *atc-io-reduction-on-sets* $M1\ T\ \Omega\ M2 =$
 $(L_{in}\ M1\ T \subseteq L_{in}\ M2\ T$
 $\wedge (\bigcup_{io \in L_{in}} M1\ T. \{io\} \times B\ M1\ io\ \Omega)$
 $\subseteq (\bigcup_{io \in L_{in}} M2\ T. \{io\} \times B\ M2\ io\ \Omega))$
 ⟨*proof*⟩

lemma *asc-algorithm-correctness*:
VARs $tsN\ cN\ rmN\ obs\ obsI\ obs\Omega\ obsI\Omega\ iter\ isReduction$
 $\{$
 $\quad OFSM\ M1 \wedge OFSM\ M2 \wedge \text{asc-fault-domain } M2\ M1\ m \wedge \text{test-tools } M2\ M1\ FAIL\ PM\ V\ \Omega$
 $\}$
 $tsN := \{\};$
 $cN := V;$
 $rmN := \{\};$
 $obs := L_{in}\ M2\ cN;$
 $obsI := L_{in}\ M1\ cN;$
 $obs\Omega := (\bigcup_{io \in L_{in}} M2\ cN. \{io\} \times B\ M2\ io\ \Omega);$
 $obsI\Omega := (\bigcup_{io \in L_{in}} M1\ cN. \{io\} \times B\ M1\ io\ \Omega);$
 $iter := 1;$
 $WHILE\ (cN \neq \{\}) \wedge obsI \subseteq obs \wedge obsI\Omega \subseteq obs\Omega$
 $INV\ \{$
 $\quad 0 < iter$
 $\quad \wedge tsN = TS\ M2\ M1\ \Omega\ V\ m\ (iter-1)$
 $\quad \wedge cN = C\ M2\ M1\ \Omega\ V\ m\ iter$
 $\quad \wedge rmN = RM\ M2\ M1\ \Omega\ V\ m\ (iter-1)$
 $\quad \wedge obs = L_{in}\ M2\ (tsN \cup cN)$
 $\quad \wedge obsI = L_{in}\ M1\ (tsN \cup cN)$
 $\quad \wedge obs\Omega = (\bigcup_{io \in L_{in}} M2\ (tsN \cup cN). \{io\} \times B\ M2\ io\ \Omega)$

```

 $\wedge \text{obsI}_\Omega = (\bigcup_{io \in L_{in} M1} (tsN \cup cN). \{io\} \times B M1 io \Omega)$ 
 $\wedge \text{OFSM } M1 \wedge \text{OFSM } M2 \wedge \text{asc-fault-domain } M2 M1 m \wedge \text{test-tools } M2 M1 \text{ FAIL } PM V \Omega$ 
}
DO
  iter := iter + 1;
  rmN := {xs' ∈ cN .
    (¬ (Lin M1 {xs'} ⊆ Lin M2 {xs'}))
    ∨ (∀ io ∈ Lin M1 {xs'} .
      (∃ V'' ∈ N io M1 V .
        (∃ S1 .
          (∃ vs xs .
            io = (vs@xs)
            ∧ mcp (vs@xs) V'' vs
            ∧ S1 ⊆ nodes M2
            ∧ (∀ s1 ∈ S1 . ∀ s2 ∈ S1 .
              s1 ≠ s2 →
                (∀ io1 ∈ RP M2 s1 vs xs V'' .
                  ∀ io2 ∈ RP M2 s2 vs xs V'' .
                    B M1 io1 Ω ≠ B M1 io2 Ω))
                ∧ m < LB M2 M1 vs xs (tsN ∪ V) S1 Ω V'' ))))));
  tsN := tsN ∪ cN;
  cN := append-set (cN - rmN) (inputs M2) - tsN;
  obs := obs ∪ Lin M2 cN;
  obsI := obsI ∪ Lin M1 cN;
  obsΩ := obsΩ ∪ (∪io ∈ Lin M2 cN. {io} × B M2 io Ω);
  obsIΩ := obsIΩ ∪ (∪io ∈ Lin M1 cN. {io} × B M1 io Ω)
OD;
isReduction := ((obsI ⊆ obs) ∧ (obsIΩ ⊆ obsΩ))
{
  isReduction = M1 ≲ M2 — variable isReduction is used only as a return value, it is true if and only if M1 is a
reduction of M2
}
⟨proof⟩

```

```

end
theory ASC-Example
  imports ASC-Hoare
begin

```

8 Example product machines and properties

This section provides example FSMs and shows that the assumptions on the inputs of the adaptive state counting algorithm are not vacuous.

8.1 Constructing FSMs from transition relations

This subsection provides a function to more easily create FSMs, only requiring a set of transition-tuples and an initial state.

```

fun from-rel :: ('state × ('in × 'out) × 'state) set ⇒ 'state ⇒ ('in, 'out, 'state) FSM where
from-rel rel q0 = (| succ = λ io p . { q . (p,io,q) ∈ rel },
  inputs = image (fst ∘ fst ∘ snd) rel,
  outputs = image (snd ∘ fst ∘ snd) rel,
  initial = q0 |)

```

```

lemma nodes-from-rel : nodes (from-rel rel q0) ⊆ insert q0 (image (snd ∘ snd) rel)
  (is nodes ?M ⊆ insert q0 (image (snd ∘ snd) rel))
⟨proof⟩

```

```

fun well-formed-rel :: ('state × ('in × 'out) × 'state) set ⇒ bool where

```

$well\text{-formed}\text{-rel } rel = (finite \text{ rel}$
 $\wedge (\forall s1 \ x \ y . (x \notin image \ (fst \circ fst \circ snd) \ rel$
 $\vee y \notin image \ (snd \circ fst \circ snd) \ rel)$
 $\longrightarrow \neg(\exists s2 . (s1, (x, y), s2) \in rel))$
 $\wedge rel \neq \{\})$

lemma *well-formed-from-rel* :
assumes *well-formed-rel rel*
shows *well-formed (from-rel rel q0) (is well-formed ?M)*
 $\langle proof \rangle$

fun *completely-specified-rel-over* :: ('state \times ('in \times 'out) \times 'state) set \Rightarrow 'state set \Rightarrow bool
where
completely-specified-rel-over rel nods = ($\forall s1 \in nods .$
 $\forall x \in image \ (fst \circ fst \circ snd) \ rel .$
 $\exists y \in image \ (snd \circ fst \circ snd) \ rel .$
 $\exists s2 . (s1, (x, y), s2) \in rel$)

lemma *completely-specified-from-rel* :
assumes *completely-specified-rel-over rel (nodes ((from-rel rel q0)))*
shows *completely-specified (from-rel rel q0) (is completely-specified ?M)*
 $\langle proof \rangle$

fun *observable-rel* :: ('state \times ('in \times 'out) \times 'state) set \Rightarrow bool **where**
observable-rel rel = ($\forall io \ s1 . \{ s2 . (s1, io, s2) \in rel \} = \{\}$
 $\vee (\exists s2 . \{ s2' . (s1, io, s2') \in rel \} = \{s2\}))$)

lemma *observable-from-rel* :
assumes *observable-rel rel*
shows *observable (from-rel rel q0) (is observable ?M)*
 $\langle proof \rangle$

abbreviation *OFMSM-rel rel q0* \equiv *well-formed-rel rel*
 \wedge *completely-specified-rel-over rel (nodes (from-rel rel q0))*
 \wedge *observable-rel rel*

lemma *OFMSM-from-rel* :
assumes *OFMSM-rel rel q0*
shows *OFMSM (from-rel rel q0)*
 $\langle proof \rangle$

8.2 Example FSMs and properties

abbreviation *M_S-rel* :: (nat \times (nat \times nat) \times nat) set \equiv $\{(0, (0, 0), 1), (0, (0, 1), 1), (1, (0, 2), 1)\}$
abbreviation *M_S* :: (nat, nat, nat) FSM \equiv *from-rel M_S-rel 0*

abbreviation *M_I-rel* :: (nat \times (nat \times nat) \times nat) set \equiv $\{(0, (0, 0), 1), (0, (0, 1), 1), (1, (0, 2), 0)\}$
abbreviation *M_I* :: (nat, nat, nat) FSM \equiv *from-rel M_I-rel 0*

lemma *example-nodes* :
nodes M_S = {0, 1} *nodes M_I = {0, 1}*
 $\langle proof \rangle$

lemma *example-OFSM* :
 OFSM M_S OFSM M_I
 ⟨proof⟩

lemma *example-fault-domain* : *asc-fault-domain* M_S M_I 2
 ⟨proof⟩

abbreviation $FAIL_I :: (nat \times nat) \equiv (3,3)$
abbreviation $PM_I :: (nat, nat, nat \times nat) FSM \equiv ()$
 $succ = (\lambda a (p1,p2) . (if (p1 \in nodes\ M_S \wedge p2 \in nodes\ M_I \wedge (fst\ a \in inputs\ M_S)$
 $\wedge (snd\ a \in outputs\ M_S \cup outputs\ M_I))$
 $then (if (succ\ M_S\ a\ p1 = \{\}) \wedge succ\ M_I\ a\ p2 \neq \{\})$
 $then \{FAIL_I\}$
 $else (succ\ M_S\ a\ p1 \times succ\ M_I\ a\ p2))$
 $else \{\}))$,
 $inputs = inputs\ M_S$,
 $outputs = outputs\ M_S \cup outputs\ M_I$,
 $initial = (initial\ M_S, initial\ M_I)$
 ⟩

lemma *example-productF* : *productF* M_S M_I $FAIL_I$ PM_I
 ⟨proof⟩

abbreviation $V_I :: nat\ list\ set \equiv \{\[], [0]\}$

lemma *example-det-state-cover* : *is-det-state-cover* M_S V_I
 ⟨proof⟩

abbreviation $\Omega_I :: (nat, nat) ATC\ set \equiv \{ Node\ 0\ (\lambda y . Leaf) \}$

lemma *applicable-set* M_S Ω_I
 ⟨proof⟩

lemma *example-test-tools* : *test-tools* M_S M_I $FAIL_I$ PM_I V_I Ω_I
 ⟨proof⟩

lemma *OFSM-not-vacuous* :
 $\exists M :: (nat, nat, nat) FSM . OFSM\ M$
 ⟨proof⟩

lemma *fault-domain-not-vacuous* :
 $\exists (M2 :: (nat, nat, nat) FSM) (M1 :: (nat, nat, nat) FSM) m . asc-fault-domain\ M2\ M1\ m$
 ⟨proof⟩

lemma *test-tools-not-vacuous* :
 $\exists (M2 :: (nat, nat, nat) FSM)$
 $(M1 :: (nat, nat, nat) FSM)$
 $(FAIL :: (nat \times nat))$
 $(PM :: (nat, nat, nat \times nat) FSM)$
 $(V :: (nat\ list\ set))$
 $(\Omega :: (nat, nat) ATC\ set) . test-tools\ M2\ M1\ FAIL\ PM\ V\ \Omega$
 ⟨proof⟩

lemma *precondition-not-vacuous* :

```

shows  $\exists (M2::(nat,nat,nat) FSM)$ 
   $(M1::(nat,nat,nat) FSM)$ 
   $(FAIL::(nat \times nat))$ 
   $(PM::(nat,nat,nat \times nat) FSM)$ 
   $(V::(nat list set))$ 
   $(\Omega::(nat,nat) ATC set)$ 
   $(m :: nat) .$ 
   $OFSM M1 \wedge OFSM M2 \wedge asc-fault-domain M2 M1 m \wedge test-tools M2 M1 FAIL PM V \Omega$ 
 $\langle proof \rangle$ 
end

```

References

- [1] J. Brunner. Transition systems and automata. *Archive of Formal Proofs*, Oct. 2017. http://isa-afp.org/entries/Transition_Systems_and_Automata.html, Formal proof development.
- [2] R. M. Hierons. Testing from a nondeterministic finite state machine using adaptive state counting. *IEEE Transactions on Computers*, 53(10):1330–1342, 2004.
- [3] R. Sachtleben, J. Peleska, R. Hierons, and W.-L. Huang. A mechanised proof of an adaptive state counting algorithm. In *IFIP International Conference on Testing Software and Systems*. Springer, 2019. to appear.