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
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 \texttt{well\_formed}, together with the finiteness of the state set.

\begin{verbatim}
record ('in, 'out, 'state) FSM =
  suc, :: ('in × 'out) ⇒ 'state ⇒ '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 \cite{1}.

\begin{verbatim}
global-interpretation FSM : transition-system-initial
  λ a p. snd a — execute
  λ a p. snd a ∈ suc A (fst a) p — enabled
  λ p. p = initial A — initial
  for A
    defines path = FSM.path
    and run = FSM.run
    and reachable = FSMreachable
    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 \cite{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
\end{verbatim}

\end{verbatim}
abbreviation \( \text{trace} \equiv \text{FSM.trace} \)

abbreviation \( \text{successors} :: (\text{in}, \text{out}, \text{state}, \text{more}) \) \( \text{FSM-scheme} \Rightarrow \text{state} \Rightarrow \text{state set} \) where
\( \text{successors} \equiv \text{FSM.successors} \) \( \text{TYPE}(\text{in}) \) \( \text{TYPE}(\text{out}) \) \( \text{TYPE}(\text{more}) \)

lemma \( \text{states-alt-def} : \text{states } r \ p = \text{map snd } r \)
\( \langle \text{proof} \rangle \)

lemma \( \text{trace-alt-def} : \text{trace } r \ p = \text{smap snd } r \)
\( \langle \text{proof} \rangle \)

definition \( \text{language-state} :: (\text{in}, \text{out}, \text{state}) \) \( \text{FSM} \Rightarrow \text{state} \Rightarrow (\text{in} \times \text{out}) \) \( \text{list set} \) \( \text{(LS)} \)
where
\( \text{language-state } M \ q \equiv \{ \text{map fst } r \mid r. \text{path } M \ r \ q \} \)

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

abbreviation \( L \ M \equiv \text{LS } M \ ) \ (\text{initial } M) \)

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

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

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

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

lemma \( \text{language-state-prefix} : \)
assumes \( w1 @ w2 \in \text{LS } M \ q \)
shows \( w1 \in \text{LS } M \ q \)
\( \langle \text{proof} \rangle \)

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

lemma \( \text{states-target-index} : \)
assumes \( i < \text{length } p \)
shows \( \text{states } p \ q1 \mid i = \text{target } (\text{take } (\text{Suc } i) \ p) \ q1 \)
\( \langle \text{proof} \rangle \)

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 ⇒ ('in, 'out, 'state2) FSM where
product A B ≡
| succ = λ a (p₁, p₂). succ A a p₁ × succ B a p₂,
| inputs = inputs A ∪ inputs B,
| outputs = outputs A ∪ outputs B,
| initial = (initial A, initial B)
|

lemma product-simps[simp]:
succ (product A B) a (p₁, p₂) = succ A a p₁ × succ B a p₂
inputs (product A B) = inputs A ∪ inputs B
outputs (product A B) = outputs A ∪ outputs B
initial (product A B) = (initial A, initial B)
(proof)

lemma product-target[simp]:
assumes length w = 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 r₂
shows path (product A B) (w || r₁ || r₂) (p₁, p₂) ⟷ path A (w || r₁) p₁ ∧ path B (w || r₂) p₂
(proof)

lemma product-language-state[simp]: LS (product A B) (q₁,q₂) = LS A q₁ ∩ LS B q₂
(proof)

lemma product-nodes:
nodes (product A B) ⊆ nodes A × 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 ⇒ bool where
finite-FSM M = (finite (nodes M)
∧ finite (inputs M)
∧ finite (outputs M))

fun observable :: ('in, 'out, 'state) FSM ⇒ bool where
observable M = (∀ t . ∃ s₁ . ((succ M) t s₁ = {1})
∧ (∃ s₂ . (succ M) t s₁ = {s₂}))

fun completely-specified :: ('in, 'out, 'state) FSM ⇒ bool where
completely-specified M = (∀ s₁ ∈ nodes M . ∀ x ∈ inputs M .
∃ y ∈ outputs M .
∃ s₂ . s₂ ∈ (succ M) (x,y) s₁)

fun well-formed :: ('in, 'out, 'state) FSM ⇒ bool where
well-formed M = (finite-FSM M
∧ (∀ s₁ x y . (x /∈ inputs M ∨ y /∈ outputs M)
→ suc M (x,y) s₁ = {1})
∧ inputs M ≠ {1}
∧ outputs M ≠ {1})

abbreviation OFSM M ≡ well-formed M
∧ observable M
∧ completely-specified M

lemma OFSM-props[elim!]:
assumes OFSM M
shows well-formed M
  observable M
  completely-specified M (proof)

lemma set-of-succs-finite :
  assumes well-formed M
  and q ∈ nodes M
  shows finite (succ M io q) (proof)

lemma well-formed-path-io-containment :
  assumes well-formed M
  and path M p q
  shows set (map fst p) ⊆ (inputs M × outputs M) (proof)

lemma path-input-containment :
  assumes well-formed M
  and path M p q
  shows set (map fst (map fst p)) ⊆ inputs M (proof)

lemma path-state-containment :
  assumes path M p q
  and q ∈ nodes M
  shows set (map snd p) ⊆ nodes M (proof)

lemma language-state-inputs :
  assumes well-formed M
  and io ∈ language-state M q
  shows set (map fst io) ⊆ inputs M (proof)

lemma set-of-paths-finite :
  assumes well-formed M
  and q1 ∈ nodes M
  shows finite { p . path M p q1 ∧ target p q1 = q2 ∧ length p ≤ k } (proof)

lemma non-distinct-duplicate-indices :
  assumes ¬ distinct xs
  shows ∃ i1 i2 . i1 ≠ i2 ∧ xs ! i1 = xs ! i2 ∧ length xs ≤ length xs (proof)

lemma reaching-path-without-repetition :
  assumes well-formed M
  and q2 ∈ reachable M q1
  and q1 ∈ nodes M
  shows ∃ p . path M p q1 ∧ target p q1 = q2 ∧ distinct (q1 ≠ states p q1) (proof)

lemma observable-path-unique[simp] :
  assumes io ∈ LS M q
  and observable M
  and path M (io || tr1) q length io = length tr1
  and path M (io || tr2) q length io = length tr2
  shows tr1 = tr2
lemma observable-path-unique-ex[elim] :
  assumes observable M
  and io ∈ LS M q
  obtains tr
  where \{ t . path M (io || t) q ∧ length io = length t \} = \{ tr \}
⟨proof⟩

lemma well-formed-product[simp] :
  assumes well-formed M1
  and well-formed M2
  shows well-formed (product M2 M1) (is well-formed ?PM)
⟨proof⟩

1.5 States reached by a given IO-sequence

Function \texttt{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.

\texttt{fun io-targets :: (′in, ′out, ′state) FSM ⇒ ′state ⇒ (′in × ′out) list ⇒ ′state set where}
\texttt{io-targets M q io = \{ target (io || tr) q | tr . path M (io || tr) q ∧ length io = length tr \}}

lemma io-target-implies-L :
  assumes q ∈ io-targets M (initial M) io
  shows io ∈ L M
⟨proof⟩

lemma io-target-from-path :
  assumes path M (w || tr) q
  and length w = length tr
  shows target (w || tr) q ∈ io-targets M q w
⟨proof⟩

lemma io-targets-observable-singleton-ex :
  assumes observable M
  and io ∈ LS M q1
  obtains q2
    where io-targets M q1 io = \{ q2 \}
⟨proof⟩

lemma io-targets-observable-singleton-ob :
  assumes observable M
  and io ∈ LS M q1
  obtains q2
    where io-targets M q1 io = \{ q2 \}
⟨proof⟩

lemma io-targets-elim[elim] :
  assumes p ∈ io-targets M q io
  obtains tr
  where target (io || tr) q = p ∧ path M (io || tr) q ∧ length io = length tr
⟨proof⟩

lemma io-targets-reachable :
  assumes q2 ∈ io-targets M q1 io
  shows q2 ∈ reachable M q1
⟨proof⟩

lemma io-targets-nodes :
  assumes q2 ∈ io-targets M q1 io
  and q1 ∈ nodes M
  shows q2 ∈ nodes M
⟨proof⟩
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 < length xs 
  and path M (xs || tr) q 
  and length xs = length tr 
  and length xs > 0 
  shows (states (xs || tr) q) ! i ∈ io-targets M q (take (Suc i) xs) 
(proof)

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

lemma io-path-states-prefix : 
  assumes observable M 
  and path M (io1 || tr1) q 
  and length tr1 = length io1 
  and path M (io2 || tr2) q 
  and length tr2 = length io2 
  and prefix io1 io2 
  shows tr1 = take (length tr1) tr2 
(proof)

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

lemma observable-io-target-is-singleton[simp] : 
  assumes observable M 
  and p ∈ io-targets M q io 
  shows io-targets M q io = {p} 
(proof)

lemma observable-path-prefix : 
  assumes observable M 
  and path M (io || tr) q 
  and length io = length tr 
  and path M (ioP || trP) q 
  and length ioP = length trP 
  and prefix ioP io 
  shows trP = take (length ioP) tr 
(proof)

lemma io-targets-succ : 
  assumes q2 ∈ io-targets M q1 [xy] 
  shows q2 ∈ succ M xy q1 
(proof)

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

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

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

lemma \( d\text{-reaches-unique}\text{[\text{elim}] :} \)
\( \text{assumes } d\text{-reaches } M p xs q1 \) and \( d\text{-reaches } M p xs q2 \)
\( \text{shows } q1 = q2 \)
(\text{proof})

lemma \( d\text{-reaches-unique-cases}[\text{simp}] : \)
\( \text{assumes } d\text{-reaches } M (\text{initial } M) xs q \)
\( \text{shows } \{ q . d\text{-reaches } M (\text{initial } M) xs q \} = \{ q \} \)
(\text{proof})

lemma \( d\text{-reaches-io-target} : \)
\( \text{assumes } d\text{-reaches } M (\text{initial } M) xs q \) and \( \text{length } ys = \text{length } xs \)
\( \text{shows } \text{io-targets } M p (xs || ys) \subseteq \{ q \} \)
(\text{proof})

lemma \( d\text{-reachable-reachable} : d\text{-reachable } M p \subseteq \text{reachable } M p \)
(\text{proof})

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

lemma \( \text{det-state-cover-ass-dist} : \)
\( \text{assumes } \text{is-det-state-cover-ass } M f \) and \( s1 \in \text{d-reachable } M (\text{initial } M) \)
\( \text{and } s2 \in \text{d-reachable } M (\text{initial } M) \)
\( \text{and } s1 \neq s2 \)
\( \text{shows } \neg (\text{d-reaches } M (\text{initial } M) (f s2) s1) \)
(\text{proof})

lemma \( \text{det-state-cover-ass-diff} : \)
\( \text{assumes } \text{is-det-state-cover-ass } M f \) and \( s1 \in \text{d-reachable } M (\text{initial } M) \)
\( \text{and } s2 \in \text{d-reachable } M (\text{initial } M) \)
\( \text{and } s1 \neq s2 \)
\( \text{shows } f s1 \neq f s2 \)
(\text{proof})

fun \( \text{is-det-state-cover} :: (\text{in}, \text{out}, \text{state}) \text{FSM} \Rightarrow \text{in list set} \Rightarrow \text{bool} \) where
is-det-state-cover $M V = (\exists f . \text{is-det-state-cover-ass} M f \\
\land V = \text{image} f (d\text{-reachable} M (\text{initial} M)))$

**Lemma det-state-cover-d-reachable [elim]**

- Assumes `is-det-state-cover M V` and $v \in V$
- Obtains $q$ where $d\text{-reaches} M (\text{initial} M) v q$

**Lemma det-state-cover-card [simp]**

- Assumes `is-det-state-cover M V` and finite `(nodes M)`
- Shows $\text{card} (d\text{-reachable} M (\text{initial} M)) = \text{card} V$

**Lemma det-state-cover-finite**

- Assumes `is-det-state-cover M V` and finite `(nodes M)`
- Shows finite $V$

**Lemma det-state-cover-initial**

- Assumes `is-det-state-cover M V`
- Shows $[] \in V$

**Lemma det-state-cover-empty**

- Assumes `is-det-state-cover M V`
- Shows $[] \in V$

### 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 $\leq 200$)

- Where $M1 \preceq M2 = (LS M1 (\text{initial} M1) \subseteq LS M2 (\text{initial} M2))$

**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 $io\text{-targets} M1 q1 io = \{ q1t \}$
- And $io\text{-targets} M2 q2 io = \{ q2t \}$
- Shows $LS M1 q1t \subseteq LS M2 q2t$

### 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 $M q xs = \{(xs || ys) \mid ys . (\text{length} xs = \text{length} ys \land (xs || ys) \in LS M q)\}$

**Fun language-state-for-inputs** :: (`in`, `out`, `state`) FSM $\Rightarrow$ `state $\Rightarrow$ `in list $\Rightarrow$ (`in $\times$ `out`) list set
where
language-state-for-inputs $M \ q \ ISeqs = \{(xs \ || \ ys) \mid xs \ ys \ . \ (xs \in ISeqs$

\land \ \text{length} \ xs = \text{length} \ ys$

\land \ (xs \ || \ ys) \in LS \ M \ q\}\}$

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

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

notation $\text{io-reduction-on} \ ((\cdot \ - \ [-]) \ [1000,0,0] \ 61)$

notation $\text{io-reduction-on} \ ((\cdot \ - \ -) \ [1000,0,0] \ 61)$

lemma language-state-for-input-alt-def :
language-state-for-input $M \ q \ xs = LS_{\text{in}} \ M \ q \ \{xs\}$
(proof)

lemma language-state-for-inputs-alt-def :
$LS_{\text{in}} \ M \ q \ ISeqs = \bigcup \ (\text{image} \ (\text{language-state-for-input} \ M \ q) \ ISeqs)$
(proof)

lemma language-state-for-inputs-in-language-state :
$LS_{\text{in}} \ M \ q \ T \subseteq \text{language-state} \ M \ q$
(proof)

lemma language-state-for-inputs-map-fst :
assumes $io \in \text{language-state} \ M \ q$
and \ $\text{map} \ \text{fst} \ io \in T$
shows $io \in LS_{\text{in}} \ M \ q \ T$
(proof)

lemma language-state-for-inputs-nonempty :
assumes $\text{set} \ xs \subseteq \text{inputs} \ M$
and \ $\text{completely-specified} \ M$
and \ $q \in \text{nodes} \ M$
shows $LS_{\text{in}} \ M \ q \ \{xs\} \neq \{\}$
(proof)

lemma language-state-for-inputs-map-fst-contained :
assumes $vs \in LS_{\text{in}} \ M \ q \ V$
shows $\text{map} \ \text{fst} \ vs \in V$
(proof)

lemma language-state-for-inputs-empty :
assumes $[] \in V$
shows $[] \in LS_{\text{in}} \ M \ q \ V$
(proof)

lemma language-state-for-input-empty[simp] :
language-state-for-input $M \ q \ [] = \{[]\}$
(proof)

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

lemma language-state-for-inputs-prefix :
assumes $\text{vs} \ @ \ xs \in L_{\text{in}} \ M1 \ \{\text{vs} \ @ \ xs'\}$
and \ $\text{length} \ vs = \text{length} \ us'$
shows $vs \in L_{\text{in}} \ M1 \ \{us'\}$
(proof)

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 $M$.

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 in, out out, state FSM) ⇒ (in in, out out, state FSM) ⇒ (in in × out out) list ⇒ bool
where
sequence-to-failure M1 M2 xs = (
  (butlast xs) ∈ (language-state M2 (initial M2) ∩ language-state M1 (initial M1))
  ∧ xs ∈ (language-state M1 (initial M1) − language-state M2 (initial M2)))

lemma sequence-to-failure-ob :
  assumes ¬ M1 ⪯ 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 $∀ q ∈ io-targets M2 (initial M2) (butlast io) . succ M2 (last io) q = {}$
(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 ∈ L M1 − L M2
  and vs ∈ L M2 ∩ L M1
  shows $∃ xs'. prefix xs' xs ∧ sequence-to-failure M1 M2 (vs@xs')$
(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 ¬ M1 ⪯ M2
  shows $∃ xs . sequence-to-failure M1 M2 xs ∧ length xs ≤ |M2| * |M1|$
(proof)

1.11 Minimal sequence to failure extending

A minimal sequence to a failure extending 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 list) ⇒ (in in, out out, state FSM) ⇒ (in in, out out, state FSM) ⇒ (in in × out out) list
  ⇒ (in in × out out) list ⇒ bool
where
minimal-sequence-to-failure-extending V M1 M2 v' io = (
\[ v' \in \text{L}_{i,n} M_1 \ V \land \text{sequence-to-failure} M_1 M_2 (v' \oplus \text{io}) \land \neg (\exists \text{io}' . \exists w' \in \text{L}_{i,n} M_1 \ V . \text{sequence-to-failure} M_1 M_2 (w' \oplus \text{io}')) \land \text{length} \text{io}' < \text{length} \text{io}) \]

**Lemma** minimal-sequence-to-failure-extending-det-state-cover-ob :
- Assumes well-formed \( M_1 \) and well-formed \( M_2 \) and observable \( M_2 \) and is-det-state-cover \( M_2 \ V \) and \( \neg M_1 \leq M_2 \)
- Obtains \( vs \, xs \)
- Where minimal-sequence-to-failure-extending \( V M_1 M_2 \, vs \, xs \)

**Proof**

**Lemma** mstfe-prefix-input-in-V :
- Assumes minimal-sequence-to-failure-extending \( V M_1 M_2 \, vs \, xs \)
- Shows \((\text{map} \, \text{fst} \, vs) \in V\)

**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 \( n \times m \) where \( n \) is the number of states of the reference model (here FSM \( M_2 \)) and \( m \) is an upper bound on the number of states of the SUT (here FSM \( M_1 \)).

**Lemma** product-suite-soundness :
- Assumes well-formed \( M_1 \) and well-formed \( M_2 \) and observable \( M_1 \) and observable \( M_2 \) and \( \text{inputs} M_2 = \text{inputs} M_1 \) and \( |M_1| \leq m \)
- Shows \( \neg M_1 \leq M_2 \rightarrow \neg M_1 \leq\{\{xs . \text{set} \, xs \subseteq \text{inputs} \, M_2 \land \text{length} \, xs \leq |M_2| \times \, m\}\} M_2 \)

**Proof**

**Lemma** product-suite-completeness :
- Assumes well-formed \( M_1 \) and well-formed \( M_2 \) and observable \( M_1 \) and observable \( M_2 \) and \( \text{inputs} M_2 = \text{inputs} M_1 \) and \( |M_1| \leq m \)
- Shows \( M_1 \leq M_2 \iff M_1 \leq\{\{xs . \text{set} \, xs \subseteq \text{inputs} \, M_2 \land \text{length} \, xs \leq |M_2| \times \, m\}\} M_2 \)

**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 \( A B \) is the product machine of \( A \) and \( B \) extended with fail state FAIL.

**Fun** productF :: ('in', 'out', 'state1) FSM \Rightarrow ('in', 'out', 'state2) FSM \Rightarrow ('state1 \times \text{state2}) \\
\Rightarrow ('in', 'out', 'state1 \times \text{state2}) \, FSM \Rightarrow \text{bool} \, \text{where}
\[
\text{productF} A B \text{ FAIL AB} = \{
\begin{align*}
\text{inputs} A &= \text{inputs B} \\
\wedge (\text{fst FAIL} \notin \text{nodes A}) \\
\wedge (\text{snd FAIL} \notin \text{nodes B}) \\
\wedge AB &= \{
\begin{align*}
succ &= (\lambda a (p1,p2). (\text{if } p1 \in \text{nodes A} \wedge p2 \in \text{nodes B} \\
\wedge (\text{fst a} \in \text{inputs A}) \\
\wedge (\text{snd a} \in \text{outputs A} \cup \text{outputs B})) \\
\text{then } (\text{if } \text{succ A a p1} = \{\} \wedge \text{succ B a p2} \neq \{\}) \\
\text{then } \{\text{FAIL}\} \\
\text{else } (\text{succ A a p1} \times \text{succ B a p2}) \\
\text{else } \{\}\),
\end{align*}
\}
\}
\]
\]
shows succ AB to FAIL = {}
(proof)

lemma no-prefix-targets-FAIL :
  assumes productF M2 M1 FAIL PM
  and path PM p q
  and k < length p
  shows target (take k p) q ≠ FAIL
  (proof)

lemma productF-path-inclusion :
  assumes length w = length r1 length r1 = length r2
  and productF A B FAIL AB
  and well-formed A
  and well-formed B
  and path A (w || r1) p1 ∧ path B (w || r2) p2
  and p1 ∈ nodes A
  and p2 ∈ nodes B
  shows path (AB) (w || r1 || r2) (p1, p2)
  (proof)

lemma productF-path-forward :
  assumes length w = length r1 length r1 = length r2
  and productF A B FAIL AB
  and well-formed A
  and well-formed B
  and (path A (w || r1) p1 ∧ path B (w || r2) p2)
  ∨ (target (w || r1 || r2) (p1, p2) = FAIL
  ∧ length w > 0
  ∧ path A (butlast (w || r1)) p1
  ∧ path B (butlast (w || r2)) p2
  ∧ succ A (last w) (target (butlast (w || r1)) p1) = {}
  ∧ succ B (last w) (target (butlast (w || r2)) p2) ≠ {}
  and p1 ∈ nodes A
  and p2 ∈ nodes B
  shows path (AB) (w || r1 || r2) (p1, p2)
  (proof)

lemma butlast-zip-cons : length ws = length r1s ⇒ ws ≠ []
                             ⇒ butlast (w # ws || r1 # r1s) = ((w, r1) # (butlast (ws || r1s)))
  (proof)

lemma productF-succ-fail-imp :
  assumes productF A B FAIL AB
  and FAIL ∈ succ AB w (p1, p2)
  and well-formed A
  and well-formed B
  shows p1 ∈ nodes A ∧ p2 ∈ nodes B ∧ (fst w ∈ inputs A) ∧ (snd w ∈ outputs A ∪ outputs B)
  ∧ succ AB w (p1, p2) = {FAIL} ∧ succ A w p1 = {} ∧ succ B w p2 ≠ {}
  (proof)

lemma productF-path-reverse :
  assumes length w = length r1 length r1 = length r2
  and productF A B FAIL AB
  and well-formed A
  and well-formed B
  and path AB (w || r1 || r2) (p1, p2)
  and p1 ∈ nodes A
and \( p_2 \in \text{nodes } B \)

shows \((\text{path } A (w || r_1) \land \text{path } B (w || r_2) p_2) \lor (\text{target } (w || r_1 || r_2) (p_1, p_2) = \text{FAIL}) \land \text{length } w > 0 \land \text{path } A (\text{butlast } (w || r_1)) p_1 \land \text{path } B (\text{butlast } (w || r_2)) p_2 \land \text{succ } A (\text{last } w) (\text{target } (\text{butlast } (w || r_1)) p_1) = \{\} \land \text{succ } B (\text{last } w) (\text{target } (\text{butlast } (w || r_2)) p_2) \neq \{\})\)

⟨proof⟩

lemma \text{butlast-zip[simp]} :
assumes \text{length } xs = \text{length } ys
shows \text{butlast } (xs || ys) = (\text{butlast } xs || \text{butlast } ys)
 ⟨proof⟩

lemma \text{productF-path-reverse-ob} :
assumes \text{length } w = \text{length } r_1 \land \text{length } r_1 = \text{length } r_2
and \text{productF } A B \text{ FAIL } AB
and \text{well-formed } A
and \text{well-formed } B
and \text{path } AB (w || r_1 || r_2) (p_1, p_2)
and \text{p_1 } \in \text{nodes } A
and \text{p_2 } \in \text{nodes } B
obtains \text{r}_2'
where \text{path } B (w || r_2') p_2 \land \text{length } w = \text{length } r_2'
 ⟨proof⟩

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

lemma \text{productF-path iff} :
assumes \text{length } w = \text{length } r_1 \land \text{length } r_1 = \text{length } r_2
and \text{productF } A B \text{ FAIL } AB
and \text{well-formed } A
and \text{well-formed } B
and \text{p_1 } \in \text{nodes } A
and \text{p_2 } \in \text{nodes } B
shows \text{path } AB (w || r_1 || r_2) (p_1, p_2) \leftrightarrow ((\text{path } A (w || r_1) p_1 \land \text{path } B (w || r_2) p_2) \lor (\text{target } (w || r_1 || r_2) (p_1, p_2) = \text{FAIL}) \land \text{length } w > 0 \land \text{path } A (\text{butlast } (w || r_1)) p_1 \land \text{path } B (\text{butlast } (w || r_2)) p_2 \land \text{succ } A (\text{last } w) (\text{target } (\text{butlast } (w || r_1)) p_1) = \{\} \land \text{succ } B (\text{last } w) (\text{target } (\text{butlast } (w || r_2)) p_2) \neq \{\})) (\text{is path } \leftrightarrow \text{paths})
 ⟨proof⟩

lemma \text{path-last-succ} :
assumes \text{path } A (ws || r_1s) p_1
and \text{length r_1s = length } ws
and \text{length } ws > 0
shows \text{last } r_1s \in \text{succ } A (\text{last } ws) (\text{target } (\text{butlast } (ws || r_1s)) p_1)
 ⟨proof⟩

lemma \text{zip-last} :
assumes \text{length } r_1 > 0
and \text{length } r_1 = \text{length } r_2
shows \text{last } (r_1 || r_2) = (\text{last } r_1, \text{last } r_2)
 ⟨proof⟩

lemma \text{productF-path-reverse-ob-2} :
assumes \text{length } w = \text{length } r_1 \land \text{length } r_1 = \text{length } r_2
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 \( \text{productF} \ M_2 \ M_1 \ \text{FAIL} \ \text{PM} \)

obtains \( p \)
where \( \text{path} \ \text{PM} \ (\text{io} || \text{p}) \ (\text{initial PM}) \land \text{length} \ \text{p} = \text{length} \ \text{io} \land \text{target} \ (\text{io} || \text{p}) \ (\text{initial PM}) = \text{FAIL} \)

\( \langle \text{proof} \rangle \)

**lemma** fail-reaching:
\[
\text{assumes} \quad \neg \ M_1 \preceq M_2 \\
\text{and} \quad \text{well-formed} \ M_1 \\
\text{and} \quad \text{well-formed} \ M_2 \\
\text{and} \quad \text{productF} \ M_2 \ M_1 \ \text{FAIL} \ \text{PM} \\
\text{shows} \quad \text{FAIL} \in \text{reachable} \ \text{PM} \ (\text{initial PM}) \\
\langle \text{proof} \rangle 
\]

**lemma** fail-reaching-ob:
\[
\text{assumes} \quad \neg \ M_1 \preceq M_2 \\
\text{and} \quad \text{well-formed} \ M_1 \\
\text{and} \quad \text{well-formed} \ M_2 \\
\text{and} \quad \text{observable} \ M_2 \\
\text{and} \quad \text{productF} \ M_2 \ M_1 \ \text{FAIL} \ \text{PM} \\
\text{obtains} \quad p \\
where \quad \text{path} \ \text{PM} \ p \ (\text{initial PM}) \ \text{target} \ p \ (\text{initial PM}) = \text{FAIL} \\
\langle \text{proof} \rangle 
\]

**lemma** fail-reaching-reverse:
\[
\text{assumes} \quad \text{well-formed} \ M_1 \\
\text{and} \quad \text{well-formed} \ M_2 \\
\text{and} \quad \text{productF} \ M_2 \ M_1 \ \text{FAIL} \ \text{PM} \\
\text{and} \quad \text{FAIL} \in \text{reachable} \ \text{PM} \ (\text{initial PM}) \\
\text{and} \quad \text{observable} \ M_2 \\
\text{shows} \quad \neg \ M_1 \preceq M_2 \\
\langle \text{proof} \rangle 
\]

**lemma** fail-reaching-iff:
\[
\text{assumes} \quad \text{well-formed} \ M_1 \\
\text{and} \quad \text{well-formed} \ M_2 \\
\text{and} \quad \text{productF} \ M_2 \ M_1 \ \text{FAIL} \ \text{PM} \\
\text{and} \quad \text{observable} \ M_2 \\
\text{shows} \quad \text{FAIL} \in \text{reachable} \ \text{PM} \ (\text{initial PM}) \iff \neg \ M_1 \preceq M_2 \\
\langle \text{proof} \rangle 
\]

**lemma** reaching-path-length:
\[
\text{assumes} \quad \text{productF} \ A \ B \ \text{FAIL} \ \text{AB} \\
\text{and} \quad \text{well-formed} \ A \\
\text{and} \quad \text{well-formed} \ B \\
\text{and} \quad q_2 \in \text{reachable} \ \text{AB} \ q_1 \\
\text{and} \quad q_2 \neq \text{FAIL} \\
\text{and} \quad q_1 \in \text{nodes} \ \text{AB} \\
\text{shows} \quad \exists \ p . \ \text{path} \ \text{AB} \ p \ q_1 \land \ \text{target} \ p \ q_1 = q_2 \land \ \text{length} \ p < \text{card} \ (\text{nodes} \ A) \ast \text{card} \ (\text{nodes} \ B) \\
\langle \text{proof} \rangle 
\]

**lemma** reaching-path-fail-length:
\[
\text{assumes} \quad \text{productF} \ A \ B \ \text{FAIL} \ \text{AB} \\
\text{and} \quad \text{well-formed} \ A \\
\text{and} \quad \text{well-formed} \ B \\
\text{and} \quad q_2 \in \text{reachable} \ \text{AB} \ q_1 \\
\text{and} \quad q_1 \in \text{nodes} \ \text{AB} \\
\langle \text{proof} \rangle 
\]
shows \( \exists \ p . \ path \ AB \ p \ q1 \land target \ p \ q1 = q2 \land length \ p \leq card (nodes \ A) \ast card (nodes \ B) \) (proof)

**lemma** \textit{productF-language}:

- assumes \textit{productF A B FAIL AB}
  - and well-formed \textit{A}
  - and well-formed \textit{B}
  - and \( io \in L \ A \cap L \ B \)
- shows \( io \in L \ AB \)

(\textit{proof})

**lemma** \textit{productF-language-state-intermediate}:

- assumes \( vs \ @ \ xs \in L \ M2 \cap L \ M1 \)
  - and \textit{productF M2 M1 FAIL PM}
  - and observable \textit{M2}
  - and observable \textit{M1}
  - and well-formed \textit{M2}
  - and well-formed \textit{M1}
- obtains \( q2 \ q1 \ tr \)
  - where \( io\text{-targets PM (initial PM) vs = \{(q2,q1)\}} \)
  - \( path \ PM (xs || tr) (q2,q1) \)
  - \( length \ xs = length \ tr \)

(\textit{proof})

**lemma** \textit{sequence-to-failure-reaches-FAIL}:

- assumes \textit{sequence-to-failure M1 M2 io}
  - and \textit{OFSM M1}
  - and \textit{OFSM M2}
  - and \textit{productF M2 M1 FAIL PM}
- shows \( FAIL \in \ io\text{-targets PM (initial PM) io} \)

(\textit{proof})

**lemma** \textit{sequence-to-failure-reaches-FAIL-ob}:

- assumes \textit{sequence-to-failure M1 M2 io}
  - and \textit{OFSM M1}
  - and \textit{OFSM M2}
  - and \textit{productF M2 M1 FAIL PM}
- shows \( \ io\text{-targets PM (initial PM) io} = \{\text{FAIL}\} \)

(\textit{proof})

**lemma** \textit{sequence-to-failure-alt-def}:

- assumes \( \ io\text{-targets PM (initial PM) io} = \{\text{FAIL}\} \)
  - and \textit{OFSM M1}
  - and \textit{OFSM M2}
  - and \textit{productF M2 M1 FAIL PM}
- shows \textit{sequence-to-failure M1 M2 io}

(\textit{proof})

end

theory \textit{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 ⇒ ('in, 'out) ATC

inductive atc-reaction :: ('in, 'out, 'state) FSM ⇒ 'state ⇒ ('in, 'out) ATC
⇒ ('in × 'out) list ⇒ bool

where
leaf[intro!]: atc-reaction M q1 Leaf [] |
node[intro!]: q2 ∈ succ M (x,y) q1
⇒ atc-reaction M q2 (f y) io
⇒ 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 ≠ 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 ∈ 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 ∃ tr. path M (io ||| tr) q1 ∧ 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 ⇒ ('in, 'out) ATC ⇒ bool where
  t ∈ range f ⇒⇒ 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 ∈ range f}

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

lemma subtest-relI [intro]: t ∈ range f ⇒⇒ (t, Node x f) ∈ subtest-rel
(proof)

lemma subtest-relI' [intro]: t = f y ⇒⇒ (t, Node x f) ∈ subtest-rel
(proof)

lemma wf-subtest-rel [simp, intro]: wf subtest-rel
function inputs-atc :: ('a,'b) ATC ⇒ 'a set where
  inputs-atc Leaf = {} |
  inputs-atc (Node x f) = insert x (⋃ (image inputs-atc (range f)))

fun applicable :: ('in,'out,'state) FSM ⇒ ('in,'out) ATC ⇒ bool where
  applicable M t = (inputs-atc t ⊆ inputs M)

fun applicable-set :: ('in,'out,'state) FSM ⇒ ('in,'out) ATC set ⇒ bool where
  applicable-set M Ω = (∀ t ∈ Ω. applicable M t)

lemma applicable-subtest :
  assumes applicable M (Node x f)
  shows applicable M (f y)

lemma IO-language : IO M q t ⊆ language-state M q

lemma IO-leaf[simp] : IO M q Leaf = {[]}

lemma IO-applicable-nonempty :
  assumes applicable M t
  and completely-specified M
  and qf ∈ nodes M
  shows IO M qf t ≠ {}
assumes applicable \( M \) \( t \)
and completely-specified \( M \)
and \( r\)-dist \( M \) \( q_1 \) \( q_2 \)
and \( q_1 \in \text{nodes} \ M \)
shows \( q_1 \neq q_2 \)
(proof)

lemma \( r\)-dist-dist :
assumes applicable-set \( M \) \( \Omega \)
and completely-specified \( M \)
and \( r\)-dist-set \( M \) \( \Omega \) \( q_1 \) \( q_2 \)
and \( q_1 \in \text{nodes} \ M \)
shows \( q_1 \neq q_2 \)
(proof)

lemma \( r\)-dist-set-dist-disjoint :
assumes applicable-set \( M \) \( \Omega \)
and completely-specified \( M \)
and \( \forall \ t_1 \in T_1 . \forall \ t_2 \in T_2 . \ r\)-dist-set \( M \) \( \Omega \) \( t_1 \) \( t_2 \)
and \( T_1 \subseteq \text{nodes} \ M \)
shows \( T_1 \cap T_2 = \{ \} \)
(proof)

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.

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

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

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

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

lemma \( B\)-dist :
assumes \( \text{io-targets} M (\text{initial} M) \text{ io1} = \{ q_1 \} \)
and \( \text{io-targets} M (\text{initial} M) \text{ io2} = \{ q_2 \} \)
and \( B M \text{ io1} \Omega \neq B M \text{ io2} \Omega \)
shows \( q_1 \neq q_2 \)
(proof)

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

lemma \text{append-\text{io-B}-in-language} :
lemma append-io-B-nonempty :
  assumes applicable-set M Ω
  and completely-specified M
  and io ∈ language-state M (initial M)
  and Ω ≠ {} 
  shows append-io-B M io Ω ≠ {} 
(proof)

lemma append-io-B-prefix-in-language :
  assumes append-io-B M io Ω ≠ {} 
  shows io ∈ 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 ⇒ (′in, ′out) ATC set ⇒ bool 
  where characterizing-atc-set M Ω = (applicable-set M Ω ∧ (∀ s1 ∈ (nodes M) . ∀ s2 ∈ (nodes M) . (∃ td . r-dist M td s1 s2) ⇒ (∃ tt ∈ Ω . r-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 ⇒ 'state ⇒ ('in', 'out) ATC set ⇒ bool 
  where atc-reduction M2 s2 M1 s1 Ω = (∀ t ∈ Ω . IO M2 s2 t ⊆ IO M1 s1 t)

— r-distinguishability holds for atc-reductions

---

---

---

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 ⇒ ('in, 'out) ATC set ⇒ bool 
  where atc-io-reduction-on M1 M2 iseq Ω = (L in M1 {iseq} ⊆ L in M2 {iseq} ∧ (∀ io ∈ L in M1 {iseq} . B M1 io Ω ⊆ B M2 io Ω))

fun atc-io-reduction-on-sets :: ('in', 'out', 'state1) FSM ⇒ 'in list set ⇒ ('in, 'out) ATC set ⇒ bool 
  where atc-io-reduction-on-sets M1 TS Ω M2 = (∀ iseq ∈ TS . atc-io-reduction-on-sets M1 M2 iseq Ω)

notation
lemma io-reduction-from-atc-io-reduction :
  assumes atc-io-reduction-on-sets M1 T Ω M2
  and finite T
  shows io-reduction-on M1 T M2
⟨proof⟩
lemma atc-io-reduction-on-subset :
  assumes atc-io-reduction-on-sets M1 T Ω M2
  and T' ⊆ T
  shows atc-io-reduction-on-sets M1 T' Ω M2
⟨proof⟩
lemma atc-reaction-reduction[intro] :
  assumes ls : language-state M1 q1 ⊆ language-state M2 q2
  and el1 : q1 ∈ nodes M1
  and el2 : q2 ∈ nodes M2
  and ret : atc-reaction M1 q1 t io
  and ob2 : observable M2
  and ob1 : observable M1
  shows atc-reaction M2 q2 t io
⟨proof⟩
lemma IO-reduction :
  assumes ls : language-state M1 q1 ⊆ language-state M2 q2
  and el1 : q1 ∈ nodes M1
  and el2 : q2 ∈ nodes M2
  and ob1 : observable M1
  and ob2 : observable M2
  shows IO M1 q1 t ⊆ IO M2 q2 t
⟨proof⟩
lemma IO-set-reduction :
  assumes ls : language-state M1 q1 ⊆ language-state M2 q2
  and el1 : q1 ∈ nodes M1
  and el2 : q2 ∈ nodes M2
  and ob1 : observable M1
  and ob2 : observable M2
  shows IO-set M1 q1 Ω ⊆ IO-set M2 q2 Ω
⟨proof⟩
lemma B-reduction :
  assumes red : M1 ⪯ M2
  and ob1 : observable M1
  and ob2 : observable M2
  shows B M1 io Ω ⊆ B M2 io Ω
⟨proof⟩
lemma append-io-B-reduction :
  assumes red : M1 ⪯ M2
  and ob1 : observable M1
  and ob2 : observable M2
  shows append-io-B M1 io Ω ⊆ append-io-B M2 io Ω
⟨proof⟩
lemma atc-io-reduction-on-reduction[intro] :
  assumes red : M1 ⪯ M2
  and ob1 : observable M1
and \( ob2 : \text{observable \( M2 \) \) shows \( \text{atc-io-reduction-on \( M1 \) \( M2 \) \( \text{iseq} \) \( \Omega \) \)} \)

\[ \langle \text{proof} \rangle \]

\texttt{lemma}\atc\texttt{-io-reduction-on-sets-reduction}[\texttt{intro}] : 
\texttt{assumes}\ \texttt{red} : \( M1 \preceq M2 \) 
\texttt{and}\ \texttt{ob1} : \texttt{observable \( M1 \) \) and\ \texttt{ob2} : \texttt{observable \( M2 \) \) \) shows \( \text{atc-io-reduction-on-sets \( M1 \) \( TS \) \( \Omega \) \( M2 \) \)} \)
\[ \langle \text{proof} \rangle \]

\texttt{lemma}\atc\texttt{-io-reduction-on-sets-via-LS}_{\texttt{i}} : 
\texttt{assumes}\ \texttt{atc-io-reduction-on-sets \( M1 \) \( TS \) \( \Omega \) \( M2 \) \) shows \( (L_{i} \in M1 TS \cup \bigcup_{io \in L_{i}} M1 TS. \ B M1 io \ \Omega) \) \subseteq \( (L_{i} \in M2 TS \cup \bigcup_{io \in L_{i}} M2 TS. \ B M2 io \ \Omega) \) \}
\[ \langle \text{proof} \rangle \]

\end{theory}

\texttt{theory}\ ASC-LB
\texttt{imports} ../\ ATC/ATC ../FSM/FSM-Product
\begin{theory}

\section{The lower bound function}

This theory defines the lower bound function \( \text{LB} \) and its properties. Function \( \text{LB} \) calculates a lower bound on the number of states of some FSM in order for some sequence to not contain certain repetitions.

\subsection{Permutation function \texttt{Perm}}

Function \texttt{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.

\texttt{fun} \texttt{Perm} :: \texttt{‘in list set} \Rightarrow \texttt{‘in \times \texttt{‘out list set set}} \texttt{where} \texttt{Perm V M} = \{ \texttt{image f V} \mid f . \forall v \in V . f v \in \texttt{language-state-for-input M (initial M) v} \}

\texttt{lemma}\ \texttt{perm-empty} : 
\texttt{assumes}\ \texttt{is-det-state-cover M2 V} 
\texttt{and}\ \texttt{V’’} \in \texttt{Perm V M1} 
\texttt{shows} \[ \in \texttt{V’’} \] 
\[ \langle \text{proof} \rangle \]

\texttt{lemma}\ \texttt{perm-elem-finite} : 
\texttt{assumes}\ \texttt{is-det-state-cover M2 V} 
\texttt{and}\ \texttt{well-formed M2} 
\texttt{and}\ \texttt{V’’} \in \texttt{Perm V M1} 
\texttt{shows}\ \texttt{finite V’’} 
\[ \langle \text{proof} \rangle \]

\texttt{lemma}\ \texttt{perm-inputs} : 
\texttt{assumes}\ \texttt{V’’} \in \texttt{Perm V M} 
\texttt{and}\ \texttt{vs} \in \texttt{V’’} 
\texttt{shows}\ \texttt{map fst vs} \in \texttt{V} 
\[ \langle \text{proof} \rangle \]

\texttt{lemma}\ \texttt{perm-inputs-diff} : 
\texttt{assumes}\ \texttt{V’’} \in \texttt{Perm V M} 
\texttt{and}\ \texttt{vs1} \in \texttt{V’’} 
\texttt{and}\ \texttt{vs2} \in \texttt{V’’} 
\texttt{and}\ \texttt{vs1} \neq \texttt{vs2} 
\texttt{shows}\ \texttt{map fst vs1} \neq \texttt{map fst vs2}
(proof)

lemma perm-language :
  assumes $V'' \in \text{Perm } V M$
  and $v_\text{s} \in V''$
  shows $v_\text{s} \in L M$
  (proof)

4.2 Helper predicates

The following predicates are used to combine often repeated assumption.

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

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

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

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

lemma perm-elem :
  assumes is-det-state-cover $M2 V$
  and OFSM $M1$
  and OFSM $M2$
  and inputs $M1 = \text{inputs } M2$
  and $v_\text{s} \in V$
  and $v_\text{s}' \in \text{language-state-for-input } M1 (\text{initial } M1) v_\text{s}$
  obtains $V''$
  where $V'' \in \text{Perm } V M1 v_\text{s}' \in V''$
  (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 $R :: (\text{in } \times \text{out } \times \text{state}) \text{ FSM } \Rightarrow \text{state } \Rightarrow (\text{in } \times \text{out}) \text{ list}$
  \Rightarrow (\text{in } \times \text{out}) \text{ list } \Rightarrow (\text{in } \times \text{out}) \text{ list set}$
  where
\[ R \ M \ s \ vs \ xs = \{ \ vs @ xs' \mid xs' \neq [] \]
\[ \land \ \text{prefix} \ xs' \ xs \]
\[ \land \ s \in \text{io-targets} \ M \ (\text{initial} \ M) \ (vs @ xs') \} \]

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 \( \forall S1 \in SS . \forall S2 \in SS . S1 = S2 \lor f S1 \cap f S2 = {} \)
  and \( \text{finite} \ SS \)
  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 \(\text{observable} \ M1 \)
  and \(\text{observable} \ M2 \)
  and \(\text{well-formed} \ M1 \)
  and \(\text{well-formed} \ M2 \)
  and \(s \in \text{nodes} \ M2 \)
  and \(\text{productF} \ M2 M1 \text{ FAIL} \ PM \)
  and \(\text{io-targets} \ PM \ (\text{initial} \ PM) \ vs = \{(q2, q1)\} \)
  and \(\text{path} \ PM \ (xs || tr) \ (q2, q1) \)
  and \(\text{length} \ xs = \text{length} \ tr \)
  and \(\text{distinct} \ (\text{states} \ (xs || tr) \ (q2, q1)) \)
  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 \(\text{observable} \ M2 \)
  shows \(\text{io-targets} \ M2 \ (\text{initial} M2) \ io = \{s\} \)
(proof)

lemma R-union-card-is-suffix-length :
  assumes \(\text{OFSM} \ M2 \)
  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 \(\text{OFSM} \ M \)
  and \(\text{Suc} \ (\text{nodes} \ M) \leq m \)
  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 \(\text{OFSM} \ M \)
  and \(\text{Suc} \ (\text{card} \ (\text{nodes} \ M) * m) \leq \text{length} \ xs \)
and \( vs \in L M \)
shows \( \exists q \in \text{nodes } M . \text{card } \left( R_M q \text{ vs xs} \right) > m \)

(\text{proof})

4.4 Function RP

Function \( \text{RP} \) extends function \( \text{MR} \) by adding all elements from a set of IO-sequences that also reach the given state.

\[
\text{fun } \text{RP} :: (\text{in} \times \text{out} \times \text{state}) \text{FSM} \Rightarrow \text{state} \Rightarrow (\text{in} \times \text{out}) \text{ list set}
\]

where
\[
\text{RP } M s \text{ vs xs } V'' = R_M s \text{ vs xs } V'' \cup \{ vs' \in V''. \text{io-targets } M \text{ (initial } M \text{) vs'} = \{s\} \}
\]

\text{lemma } \text{RP-from-R}:
assumes \( \text{is-det-state-cover } M2 V \)
and \( V'' \in \text{Perm } V M1 \)
shows \( \text{RP } M2 s \text{ vs xs } V'' = R_M s \text{ vs xs } V'' \lor \exists vs' \in V''. \text{vs'} \notin R_M s \text{ vs xs } \land \text{RP } M2 s \text{ vs xs } V'' = \text{insert vs'} (R_M s \text{ vs xs}) \)

(\text{proof})

\text{lemma } \text{finite-RP} :
assumes \( \text{is-det-state-cover } M2 V \)
and \( V'' \in \text{Perm } V M1 \)
shows \( \text{finite } (\text{RP } M2 s \text{ vs xs } V'') \)

(\text{proof})

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

(\text{proof})

\text{lemma } \text{RP-state-component-2} :
assumes \( \text{io} \in (\text{RP } M2 s \text{ vs xs } V'') \)
and \( \text{observable } M2 \)
shows \( \text{io-targets } M2 \text{ (initial } M2 \text{) io} = \{s\} \)

(\text{proof})

\text{lemma } \text{RP-io-targets-split} :
assumes \( \text{(vs @ xs)} \in L M1 \cap L M2 \)
and \( \text{observable } M1 \)
and \( \text{observable } M2 \)
and \( \text{well-formed } M1 \)
and \( \text{well-formed } M2 \)
and \( \text{productF } M2 M1 \text{ FAIL } PM \)
lemma RP-io-targets-finite-M1 :
assumes (vs @ xs) ∈ L M1 ∩ L M2
and observable M1
and is-det-state-cover M2 V
and V'' ∈ Perm V M1
shows finite (∪ (image (io-targets M1 (initial M1)) (RP M2 s vs xs V'')))
(proof)

lemma RP-io-targets-finite-PM :
assumes (vs @ xs) ∈ L M1 ∩ 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'' ∈ Perm V M1
shows finite (∪ (image (io-targets PM (initial PM)) (RP M2 s vs xs V'')))
(proof)

lemma RP-union-card-is-suffix-length :
assumes OFSM M2
and io@xs ∈ L M2
and is-det-state-cover M2 V
and V'' ∈ Perm V M1
shows ⋀ q. card (R M2 q io xs) ≤ card (RP M2 q io xs V'')
sum (λ q. card (RP M2 q io xs V'')) (nodes M2) ≥ length xs
(proof)

lemma RP-state-repetition-distribution-productF :
assumes OFSM M2
and OFSM M1
and (card (nodes M2) * m) ≤ length xs
and card (nodes M1) ≤ m
and vs@xs ∈ L M2 ∩ L M1
and is-det-state-cover M2 V
and V'' ∈ Perm V M1
shows ⋀ q ∈ nodes M2. card (RP M2 q vs xs V'') > m
(proof)

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 :: ('in', 'out', 'state1) FSM ⇒ ('in', 'out', 'state2) FSM ⇒ ('in × 'out) list
⇒ ('in × 'out) list ⇒ 'in list set ⇒ 'state1 set ⇒ ('in, 'out) ATC set
⇒ ('in × 'out) list set ⇒ bool

where
Prereq M2 M1 vs xs T S Ω V'' = (finite T)
∧ (vs @ xs) ∈ L M2 ∩ L M1
∧ S ⊆ nodes M2
\begin{align*}
    & \forall s_1 \in S \land \forall s_2 \in S \land s_1 \neq s_2 \rightarrow \\
    & \quad (\forall \text{ io1} \in \text{RP} M_2 s_1 \text{ vs } xs \text{ V''} \cdot \\
    & \quad \forall \text{ io2} \in \text{RP} M_2 s_2 \text{ vs } xs \text{ V''} \cdot \\
    & \quad B M_1 \text{ io1} \Omega \neq B M_1 \text{ io2} \Omega )))
\end{align*}

fun \text{Rep-Pre} :: ('in, 'out, 'state1) FSM \Rightarrow ('in, 'out, 'state2) FSM \Rightarrow ('in \times 'out) list \\
\Rightarrow ('in \times 'out) list = \text{bool where} \\
\text{Rep-Pre} M_2 M_1 \text{ vs } xs = (\exists \text{ x}s_1 \text{ x}s_2 \cdot \text{prefix x}s_1 \text{ x}s_2 \land \text{prefix x}s_2 \text{ x}s_1 \neq \text{x}s_2) \\
\land (\exists \text{ s}_2 \cdot \text{io-targets M}_2 \text{(initial M}_2) \text{(vs } \oplus \text{ x}s_1) = \{\text{s}_2\}) \\
\land \text{io-targets M}_2 \text{(initial M}_2) \text{(vs } \oplus \text{ x}s_2) = \{\text{s}_2\})

\land (\exists \text{ s}_1 \cdot \text{io-targets M}_1 \text{(initial M}_1) \text{(vs } \oplus \text{ x}s_1) = \{\text{s}_1\}) \\
\land \text{io-targets M}_1 \text{(initial M}_1) \text{(vs } \oplus \text{ x}s_2) = \{\text{s}_1\}))

\begin{align*}
\text{fun} \text{Rep-Cov} :: ('in, 'out, 'state1) FSM \Rightarrow ('in, 'out, 'state2) FSM \Rightarrow ('in \times 'out) list set \\
\Rightarrow ('in \times 'out) list = \text{bool where} \\
\text{Rep-Cov} M_2 M_1 \text{ V'' } \text{ vs } xs = (\exists \text{ x}s' \text{ x}s' \cdot \text{x}s' \neq \text{x}s' \land \text{prefix x}s' \text{ x}s' \land \text{x}s' \in \text{ V''}) \\
\land (\exists \text{ s}_2 \cdot \text{io-targets M}_2 \text{(initial M}_2) \text{(vs } \oplus \text{ x}s') = \{\text{s}_2\}) \\
\land \text{io-targets M}_2 \text{(initial M}_2) \text{(vs } \oplus \text{ x}s') = \{\text{s}_2\})

\land (\exists \text{ s}_1 \cdot \text{io-targets M}_1 \text{(initial M}_1) \text{(vs } \oplus \text{ x}s') = \{\text{s}_1\}) \\
\land \text{io-targets M}_1 \text{(initial M}_1) \text{(vs } \oplus \text{ x}s') = \{\text{s}_1\}))
\end{align*}

\text{lemma} \text{distinctness-via-Rep-Pre} :
\text{assumes} \neg \text{Rep-Pre M}_2 M_1 \text{ vs } xs \\
\text{and} \text{product}_F M_2 M_1 \text{ FAIL PM} \\
\text{and} \text{observable M}_1 \\
\text{and} \text{observable M}_2 \\
\text{and} \text{io-targets PM} \text{(initial PM)} \text{ vs } = \{(q_2,q_1)\} \\
\text{and} \text{path PM} \text{(x}s || \text{tr}) = \{(q_2,q_1)\} \\
\text{and} \text{length x}s = \text{length tr} \\
\text{and} \text{vs } \oplus \text{ x}s \in \text{L M}_1 \cap \text{L M}_2 \\
\text{and} \text{well-formed M}_1 \\
\text{and} \text{well-formed M}_2 \\
\text{shows} \text{distinct (states x}s || \text{tr}) = \{(q_2,q_1)\}
\text{(proof)}

\text{lemma} \text{RP-count-via-Rep-Cov} :
\text{assumes} (\text{vs } \oplus \text{ x}s \in \text{L M}_1 \cap \text{L M}_2 \\
\text{and} \text{observable M}_1 \\
\text{and} \text{observable M}_2 \\
\text{and} \text{well-formed M}_1 \\
\text{and} \text{well-formed M}_2 \\
\text{and} \text{s } \in \text{nodes M}_2 \\
\text{and} \text{product}_F M_2 M_1 \text{ FAIL PM} \\
\text{and} \text{io-targets PM} \text{(initial PM)} \text{ vs } = \{(q_2,q_1)\} \\
\text{and} \text{path PM} \text{(x}s || \text{tr}) = \{(q_2,q_1)\} \\
\text{and} \text{length x}s = \text{length tr} \\
\text{and} \text{distinct (states x}s || \text{tr}) = \{(q_2,q_1)\} \\
\text{and} \text{is-det-state-cover M}_2 \text{ V} \\
\text{and} \text{V'' } \in \text{Perm V M}_1 \\
\text{and} \neg \text{Rep-Cov M}_2 M_1 \text{ V'' vs } xs \\
\text{shows} \text{card} \left(\bigcup (\text{image (io-targets M}_1 \text{(initial M}_1)) \text{(RP M}_2 \text{ s vs } xs \text{ V''})\right) = \text{card (RP M}_2 \text{ s vs } xs \text{ V''})
\text{(proof)}

\text{lemma} \text{RP-count-alt-def} :

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

\(\langle \text{proof} \rangle\)

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.

\[
\text{fun } \text{LB} :: (\text{'}in\text{', }\text{'}out\text{', }\text{'}state\text{') }\text{FSM} \Rightarrow (\text{'}in\text{', }\text{'}out\text{', }\text{'}state\text{'') }\text{FSM} \Rightarrow (\text{'}in\text{'} \times \text{'}out\text{'}) \text{ list} \Rightarrow (\text{'}in\text{'} \times \text{'}out\text{'}) \text{ list} \Rightarrow \text{'}in \text{ list set} \Rightarrow \text{'}state1 \text{ set} \Rightarrow (\text{'}in\text{'', }\text{'}out\text{'}) \text{ ATC set} \Rightarrow (\text{'}in\text{'} \times \text{'}out\text{'}) \text{ list set} \Rightarrow \text{nat}
\]

where

\[
\text{LB } M2 M1 vs xs T S \Omega V'' = \\
\bigl( \text{sum } \lambda s . \text{card } (\text{RP } M2 s vs xs V'')) S \bigr) \\
+ \text{card } (\{ B M1 xs'' \Omega \mid xs' s'' , s'' \in S \wedge xs' \in \text{RP } M2 s' vs xs V'') \})
\]

\(\text{lemma LB-count-helper-RP-disjoint-and-cards} :\)

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

\(\langle \text{proof} \rangle\)

\(\text{lemma LB-count-helper-RP-disjoint-card-M1} :\)

assumes \((vs @ xs) \in L M1 \cap L M2\)
and observable \(M1\)
and observable \(M2\)
and well-formed \(M1\)
and well-formed \(M2\)
and \(\text{productF } M2 M1 \text{ FAIL } PM\)
and \(\text{is-det-state-cover } M2 V\)
and \(V'' \in \text{Perm } V M1\)
and \(s1 \neq s2\)
shows \(\text{card } \bigcup \text{ (image } (\text{io-targets } M1 \text{ (initial } M1)) \text{ (RP } M2 s vs xs V''))) \cup \bigcup \text{ (image } (\text{io-targets } M1 \text{ (initial } M1)) \text{ (RP } M2 s vs xs V''))) = \text{card } \bigcup \text{ (image } (\text{io-targets } M1 \text{ (initial } M1)) \text{ (RP } M2 s vs xs V''))\)
lemma LB-count-helper-RP-disjoint-M1-pair:
assumes \((v s @ x) \in L M1 \cap L M2\)
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 || tr) (q2, q1)
and length xs = length tr
and \(\neg \text{Rep-Pre M2 M1 vs xs}\)
and \(\text{is-det-state-cover M2 V}\)
and \(V'' \in \text{Perm V M1}\)
and \(\neg \text{Rep-Cov M2 M1 V'' vs xs}\)
and \(\text{Prereq M2 M1 vs xs T S} \Omega V''\)
and \(s1 \neq s2\)
and \(s1 \in S\)
and \(s2 \in S\)
and applicable-set M1 \(\Omega\)
and completely-specified M1
shows \(\text{card (RP M2 s1 vs xs V'')} + \text{card (RP M2 s2 vs xs V'')}\)
\[= \text{card (\(\bigcup\) (image (io-targets M1 (initial M1)) (RP M2 s1 vs xs V'')))}\]
\[+ \text{card (\(\bigcup\) (image (io-targets M1 (initial M1)) (RP M2 s2 vs xs V'')))}\]
\[\bigcap \text{\(\bigcup\) (image (io-targets M1 (initial M1)) (RP M2 s1 vs xs V''))}\]
\[\bigcap \text{\(\bigcup\) (image (io-targets M1 (initial M1)) (RP M2 s2 vs xs V''))}\]
\[= \{\}\]
(proof)

lemma LB-count-helper-RP-card-union :
assumes observable M2
and \(s1 \neq s2\)
shows \(\text{RP M2 s1 vs xs V'' } \cap \text{RP M2 s2 vs xs V''} = \{\}\)
(proof)

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

lemma LB-count-helper-RP-card-union-sum :
assumes \((v s @ x) \in L M2 \cap L M1\)
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 \(\text{Prereq M2 M1 vs xs T S} \Omega V''\)
and \(\neg \text{Rep-Pre M2 M1 vs xs}\)
and \(\text{OFSM M2 M1 fail V' } \Omega\)
and \(\text{is-det-state-cover M2 V}\)
and \(V'' \in \text{Perm V M1}\)
and \(\neg \text{Rep-Cov M2 M1 V'' vs xs}\)
and \(\text{Prereq M2 M1 vs xs T S} \Omega V''\)
and \(s1 \neq s2\)
and \(s1 \in S\)
and \(s2 \in S\)
and applicable-set M1 \(\Omega\)
and completely-specified M1
shows \(\text{card (RP M2 s1 vs xs V'')} + \text{card (RP M2 s2 vs xs V'')}\)
\[= \text{card (\(\bigcup\) (image (io-targets M1 (initial M1)) (RP M2 s1 vs xs V'')))}\]
\[+ \text{card (\(\bigcup\) (image (io-targets M1 (initial M1)) (RP M2 s2 vs xs V'')))}\]
\[\bigcap \text{\(\bigcup\) (image (io-targets M1 (initial M1)) (RP M2 s1 vs xs V''))}\]
\[\bigcap \text{\(\bigcup\) (image (io-targets M1 (initial M1)) (RP M2 s2 vs xs V''))}\]
\[= \{\}\]
(proof)
and \( \neg \text{Rep-Cov } M2 M1 V'' \text{ vs } xs \)

**shows**

\[
\sum (\lambda s . \text{card} (\text{RP } M2 s \text{ vs } xs V'')) S
\]

\[
= \sum (\lambda s . \text{card} (\bigcup (\text{io-targets } M1 \text{ (initial } M1)) (\text{RP } M2 s \text{ vs } xs V''))) S
\]

(\text{proof})

**lemma** finite-insert-card :

**assumes** finite (\( \bigcup SS \))

and finite \( S \)

and \( S \cap (\bigcup SS) = \{\} \)

**shows**

\[
\text{card} (\bigcup (\text{insert } S SS)) = \text{card} (\bigcup SS) + \text{card } S
\]

(\text{proof})

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

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

and \(\text{OFSM } M1\)

and \(\text{OFSM } M2\)

and \(\text{asc-fault-domain } M2 M1 m\)

and \(\text{test-tools } M2 M1 \text{ FAIL } PM V \Omega\)

and \(V'' \in \text{Perm } V M1\)

and \(\text{Prereq } M2 M1 \text{ vs } xs T S \Omega V''\)

and \(\neg \text{Rep-Pre } M2 M1 \text{ vs } xs\)

and \(\neg \text{Rep-Cov } M2 M1 V'' \text{ vs } xs\)

**shows**

\[
\sum (\lambda s . \text{card} (\text{RP } M2 s \text{ vs } xs V'')) S
\]

\[
= \text{card} (\bigcup (\lambda s . \text{image} (\lambda s . \bigcup (\text{io-targets } M1 \text{ (initial } M1)) (\text{RP } M2 s \text{ vs } xs V''))) S)
\]

(\text{proof})

**lemma** LB-count-helper-LB1 :

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

and \(\text{OFSM } M1\)

and \(\text{OFSM } M2\)

and \(\text{asc-fault-domain } M2 M1 m\)

and \(\text{test-tools } M2 M1 \text{ FAIL } PM V \Omega\)

and \(V'' \in \text{Perm } V M1\)

and \(\text{Prereq } M2 M1 \text{ vs } xs T S \Omega V''\)

and \(\neg \text{Rep-Pre } M2 M1 \text{ vs } xs\)

and \(\neg \text{Rep-Cov } M2 M1 V'' \text{ vs } xs\)

**shows**

\[
\sum (\lambda s . \text{card} (\text{RP } M2 s \text{ vs } xs V'')) S \leq \text{card} (\text{nodes } M1)
\]

(\text{proof})

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

**assumes** \(\text{observable } M\)

and \(RS \in (D M T \Omega)\)

obtains \(q\)

where \(q \in \text{nodes } M \land RS = \text{IO-set } M q \Omega\)

(\text{proof})

**lemma** LB-count-helper-LB2 :

**assumes** \(\text{observable } M1\)

and \(\text{IO-set } M1 q \Omega \in (D M1 T \Omega) \setminus \{B M1 xs' \Omega \ | \ xs' s' . s' \in S \land xs' \in \text{RP } M2 s' \text{ vs } xs V''\}\)

**shows** \(q \notin (\bigcup (\lambda s . \text{image} (\lambda s . \bigcup (\text{io-targets } M1 \text{ (initial } M1)) (\text{RP } M2 s \text{ vs } xs V''))) S))\)

(\text{proof})

4.7 Validity of the result of LB constituting a lower bound

**lemma** LB-count :

**assumes** \((vs @ xs) \in L M1\)

and \(\text{OFSM } M1\)
and OFSM M2
and asc-fault-domain M2 M1 m
and test-tools M2 M1 FAIL PM V Ω
and V'' ∈ Perm V M1
and Prereq M2 M1 vs xs T S Ω V''
and ¬ Rep-Pre M2 M1 vs xs
and ¬ Rep-Cov M2 M1 V'' vs xs
shows LB M2 M1 vs xs T S Ω V'' ≤ |M1|
(proof)

lemma contradiction-via-LB :
assumes (vs @ xs) ∈ L M1
and OFSM M1
and OFSM M2
and asc-fault-domain M2 M1 m
and test-tools M2 M1 FAIL PM V Ω
and V'' ∈ Perm V M1
and Prereq M2 M1 vs xs T S Ω V''
and ¬ Rep-Pre M2 M1 vs xs
and ¬ Rep-Cov M2 M1 V'' vs xs
and LB M2 M1 vs xs T S Ω 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 ⇒ 'a list set ⇒ 'a list ⇒ bool where
mcp z W p = (prefix p z ∧ p ∈ W ∧
(∀ p'. (prefix p' z ∧ p' ∈ W) −→ length p' ≤ length p))

lemma mcp-ex :
assumes [] ∈ 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 ⇒ 'a list set ⇒ '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 \( \text{mcp} (\text{vs} @ \text{xs'}) \) \( \text{V} \) \( \text{vs} \)
(proof)

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

5.2 Function \( \text{N} \)

Function \( \text{N} \) narrows the sets of reaction to the determinisitc 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 \( \text{N} \) :: \( \text{('} \text{in} \times \text{'} \text{out}) \text{ list} \Rightarrow \text{'} \text{in} \times \text{'} \text{out} \times \text{'} \text{state} \Rightarrow \text{FSM} \Rightarrow \text{'} \text{in} \text{ list set} \Rightarrow \text{'} \text{in} \times \text{'} \text{out} \text{ list set set} \)
where
\( \text{N} \) \( \text{io} \) \( \text{M} \) \( \text{V} \) = \{ \text{V''} \in \text{Perm} \text{V} \text{M}. (\text{map} \ \text{fst} (\text{mcp'} \text{io} \text{V''})) = (\text{mcp'} (\text{map} \ \text{fst} \text{io}) \text{V}) \}

lemma N-nonempty :
  assumes is-det-state-cover \( \text{M2} \) \( \text{V} \)
  and OFSM \( \text{M1} \)
  and OFSM \( \text{M2} \)
  and asc-fault-domain \( \text{M2} \) \( \text{M1} \) \( \text{m} \)
  and \( \text{io} \in \text{L} \text{M1} \)
shows \( \text{N} \) \( \text{io} \) \( \text{M1} \) \( \text{V} \) \( \neq \) \{\}
(proof)

lemma N-mcp-prefix :
  assumes \( \text{map} \ \text{fst} \) \( \text{vs} = \text{mcp'} (\text{map} \ \text{fst} (\text{vs} @ \text{xs})) \) \( \text{V} \)
  and \( \text{V''} \in \text{N} (\text{vs} @ \text{xs}) \) \( \text{M1} \) \( \text{V} \)
  and is-det-state-cover \( \text{M2} \) \( \text{V} \)
  and well-formed \( \text{M2} \)
  and finite \( \text{V} \)
shows \( \text{vs} \in \text{V''} \) \( \text{vs} = \text{mcp'} (\text{vs} @ \text{xs}) \) \( \text{V''} \)
(proof)

5.3 Functions \( \text{TS}, \text{C}, \text{RM} \)

Function \( \text{TS} \) 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 \( \text{TS}, \text{C} \) and \( \text{RM} \) where \( \text{TS} \) represents the test suite calculated up to some iteration, \( \text{C} \) contains the sequences considered for extension in some iteration, and \( \text{RM} \) contains the sequences of the corresponding \( \text{C} \) result that are not to be extended, which we also call removed sequences.

abbreviation append-set :: \( \text{'} \text{a list set} \Rightarrow \text{'} \text{a set} \Rightarrow \text{'} \text{a list set} \) where
append-set \( \text{T} \) \( \text{X} \) ≡ \( \{ \text{xs @ [x]} | \text{xs x . xs} \in \text{T} \land x \in \text{X} \} \)

abbreviation append-sets :: \( \text{'} \text{a list set} \Rightarrow \text{'} \text{a list set} \Rightarrow \text{'} \text{a list set} \) where
append-sets \( \text{T} \) \( \text{X} \) ≡ \( \{ \text{xs @ xs'} | \text{xs xs'} . \text{xs} \in \text{T} \land \text{xs'} \in \text{X} \} \)

fun \( \text{TS} \) :: \( \text{'} \text{in} \times \text{'} \text{out} \times \text{'} \text{state1} \) \( \text{FSM} \Rightarrow \text{'} \text{in} \times \text{'} \text{out} \times \text{'} \text{state2} \) \( \text{FSM} \Rightarrow \text{'} \text{in} \text{ list set} \Rightarrow \text{'} \text{in} \text{ list set} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{'} \text{in} \text{ list set} \)
where
\( \text{TS} \) \( \text{M2} \) \( \text{M1} \) \( \text{V} \) \( \text{m} \) \( \text{0} \) = \{\}
\( \text{TS} \) \( \text{M2} \) \( \text{M1} \) \( \text{V} \) \( \text{m} \) \( \text{0} \) = \{\}

35
shows $C M2 M1 \Omega V m i \cap C M2 M1 \Omega V m j = \{\}$
(proof)

lemma RM-subset : $RM M2 M1 \Omega V m i \subseteq C M2 M1 \Omega V m i$
(proof)

lemma RM-disj :
  assumes $i \leq j$
  and $0 < i$
  shows $RM M2 M1 \Omega V m i \cap RM M2 M1 \Omega V m (Suc j) = \{\}$
(proof)

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

lemma append-set-prefix :
  assumes $xs \in append-set T X$
  shows $butlast xs \in T$
(proof)

lemma C-subset : $C M2 M1 \Omega V m i \subseteq TS M2 M1 \Omega V m i$
(proof)

lemma TS-subset :
  assumes $i \leq j$
  shows $TS M2 M1 \Omega V m i \subseteq TS M2 M1 \Omega V m j$
(proof)

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

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 $vs@(butlast xs) \in TS M2 M1 \Omega V m i$
(proof)

lemma TS-prefix-containment :
  assumes $vs@xs \in TS M2 M1 \Omega V m i$
  and $mcp (vs@xs) V vs$
  and $prefix xs' zs$
  shows $vs@zs \in TS M2 M1 \Omega V m i$
  — 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
(proof)
lemma C-index:
assumes \( vs \not\in C M2 M1 \Omega V m i \)
and \( mcp (vs@xs) V vs \)
shows \( \text{Suc (length } xs) = i \)
(proof)

lemma TS-index:
assumes \( vs \not\in TS M2 M1 \Omega V m i \)
and \( mcp (vs@xs) V vs \)
shows \( \text{Suc (length } xs) \leq i \)
(proof)

lemma C-extension-options:
assumes \( vs \not\in C M2 M1 \Omega V m i \)
and \( mcp (vs@xs \ @ x) V vs \)
and \( x \in \text{inputs } M2 \)
and \( 0 < i \)
shows \( vs@xs @ x \not\in C M2 M1 \Omega V m (\text{Suc } i) \lor vs@xs \in RM M2 M1 \Omega V m i \)
(proof)

lemma TS-non-containment-causes:
assumes \( vs@xs \not\in TS M2 M1 \Omega V m i \)
and \( mcp (vs@xs) V vs \)
and \( \text{set } xs \subseteq \text{inputs } M2 \)
and \( 0 < i \)
shows \( (\exists xr j. xr \neq xs \land \text{prefix } xr xs \land j \leq i \land vs@xr \in RM M2 M1 \Omega V m j) \)
\lor \( (\exists xc. xc \neq xs \land \text{prefix } xc xs \land vs@xc \in (C M2 M1 \Omega V m i) \land \text{RM } M2 M1 \Omega V m i)) \)
\( (\exists) \text{PrefPreviouslyRemoved } \lor \text{PrefJustContained} \)
\( (\exists) \text{PrefPreviouslyRemoved } \lor \text{PrefJustContained} \)
\( (\exists) \text{PrefPreviouslyRemoved } \lor \text{PrefJustContained} \)

lemma TS-non-containment-causes-rev:
assumes \( mcp (vs@xs) V vs \)
and \( (\exists xr j. xr \neq xs \land \text{prefix } xr xs \land j \leq i \land vs@xr \in RM M2 M1 \Omega V m j) \)
\lor \( (\exists xc. xc \neq xs \land \text{prefix } xc xs \land vs@xc \in (C M2 M1 \Omega V m i) \land \text{RM } M2 M1 \Omega V m i)) \)
\( (\exists) \text{PrefPreviouslyRemoved } \lor \text{PrefJustContained} \)
shows \( vs@xs \not\in TS M2 M1 \Omega V m i \)
(proof)

lemma TS-finite:
assumes \( \text{finite } V \)
and \( \text{finite } (\text{inputs } M2) \)
shows \( \text{finite } (TS M2 M1 \Omega V m n) \)
(proof)
lemma C-finite :
  assumes finite V
  and finite (inputs M2)
  shows finite (C M2 M1 Ω V m n)
(proof)

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 Ω V m i ≡ TS M2 M1 Ω V m i = TS M2 M1 Ω 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 Ω
  shows final-iteration M2 M1 Ω V m i (Suc (|M2| * m))
(proof)

lemma TS-non-containment-causes-final :
  assumes vs@xs \notin TS M2 M1 Ω V m i
  and mcp (vs@xs) V vs
  and set xs \subseteq inputs M2
  and final-iteration M2 M1 Ω V m i
  and OFSM M2
  shows (∃ xr j . xr \neq xs
            ∧ prefix xr xs
            ∧ j ≤ i
            ∧ vs@xr \in RM M2 M1 Ω V m j)
(proof)

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

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 \( M2 M1 FAIL PM V \)
and \( V'' \in N (vs@xs') M1 V \)
and prefix \( zs' xs \)
shows \( \neg \text{Rep-Pre } M2 M1 vs xs' \)
(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 \( M2 M1 FAIL PM V \)
and \( V'' \in N (vs@xsR) M1 V \)
and prefix \( zsR xs \)
shows \( \neg \text{Rep-Cov } M2 M1 V'' vs xsR \)
(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 \( M2 M1 FAIL PM V \)
and \( V'' \in N (vs@xs') M1 V \)
and prefix \( zs' xs \)
shows \( \neg \text{Rep-Pre } M2 M1 vs xs' \)
and \( \neg \text{Rep-Cov } M2 M1 V'' vs xs' \)
(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 \( M2 M1 FAIL PM V \)
and final-iteration \( M2 M1 \Omega V m i \)
shows \( M1 \preceq ((TS M2 M1 \Omega V m i) \cdot \Omega) \Rightarrow M2 \rightarrow M1 \preceq M2 \)
(proof)
6.3 Main result

The following lemmata add to the previous result to show that some FSM $M_1$ is a reduction of FSM $M_2$ 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 $M_1$  
and OFSM $M_2$  
shows $M_1 \preceq M_2 \rightarrow \text{atc-io-reduction-on-sets } M_1 T \Omega M_2$  
(proof)

**Lemma asc-main-theorem**:  
assumes OFSM $M_1$  
and OFSM $M_2$  
and asc-fault-domain $M_2 M_1 m$  
and test-tools $M_2 M_1 \text{ FAIL } PM V \Omega$  
and final-iteration $M_2 M_1 \Omega V m i$  
shows $M_1 \preceq M_2 \leftarrow \rightarrow \text{atc-io-reduction-on-sets } M_1 (TS M_2 M_1 \Omega V m i) \Omega M_2$  
(proof)

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 $M_1$ is a reduction of input FSM $M_2$.

**Lemma atc-io-reduction-on-sets-from-obs**:  
assumes $L_n M_1 T \subseteq L_n M_2 T$  
and $(\bigcup_{io \in L_n} M_1 T \cdot \{io\} \times B M_1 io \Omega) \subseteq (\bigcup_{io \in L_n} M_2 T \cdot \{io\} \times B M_2 io \Omega)$  
shows atc-io-reduction-on-sets $M_1 T \Omega M_2$  
(proof)

**Lemma atc-io-reduction-on-sets-to-obs**:  
assumes atc-io-reduction-on-sets $M_1 T \Omega M_2$  
shows $L_n M_1 T \subseteq L_n M_2 T$  
and $(\bigcup_{io \in L_n} M_1 T \cdot \{io\} \times B M_1 io \Omega) \subseteq (\bigcup_{io \in L_n} M_2 T \cdot \{io\} \times B M_2 io \Omega)$  
(proof)

**Lemma atc-io-reduction-on-sets-alt-def**:  
shows atc-io-reduction-on-sets $M_1 T \Omega M_2$ = $L_n M_1 T \subseteq L_n M_2 T$  
and $(\bigcup_{io \in L_n} M_1 T \cdot \{io\} \times B M_1 io \Omega) \subseteq (\bigcup_{io \in L_n} M_2 T \cdot \{io\} \times B M_2 io \Omega)$  
(proof)

**Lemma asc-algorithm-correctness**:  
VARS $tsN$ $cN$ $rmN$ $obsN obsI obsI_\Omega$ iter isReduction  
\{  
OFSM $M_1$ $\land$ OFSM $M_2$ $\land$ asc-fault-domain $M_2 M_1 m$ $\land$ test-tools $M_2 M_1 \text{ FAIL } PM V \Omega$  
\}
\[tsN := \{\};\]
\[cN := V;\]
\[rmN := \{\};\]
\[obs := L_n \cdot M_2 \cdot cN;\]
\[obsI := L_n \cdot M_1 \cdot cN;\]
\[obs \Omega := (\bigcup \text{io} \in L_n, M_2 \cdot cN. \{\text{io}\} \times B \cdot M_2 \cdot \text{io} \cdot \Omega);\]
\[obsI \Omega := (\bigcup \text{io} \in L_n, M_1 \cdot cN. \{\text{io}\} \times B \cdot M_1 \cdot \text{io} \cdot \Omega);\]
\[iter := 1;\]
\[\text{WHILE} (cN \neq \{\} \land obs \subseteq obs \land obsI \subseteq obsI \Omega)\]
\[\text{INV} \{\]
\[\emptyset < iter\]
\[\land tsN = TS \cdot M_2 \cdot M_1 \cdot \Omega \cdot V \cdot m \ (iter - 1)\]
\[\land cN = C \cdot M_2 \cdot M_1 \cdot \Omega \cdot V \cdot m \ iter\]
\[\land rmN = RM \cdot M_2 \cdot M_1 \cdot \Omega \cdot V \cdot m \ (iter - 1)\]
\[\land obs = L_n \cdot M_2 \ (tsN \cup cN)\]
\[\land obsI = L_n \cdot M_1 \ (tsN \cup cN)\]
\[\land obsI \Omega = (\bigcup \text{io} \in L_n, M_1 \ (tsN \cup cN). \{\text{io}\} \times B \cdot M_1 \cdot \text{io} \cdot \Omega)\]
\[\land OFSM \cdot M_1 \land OFSM \cdot M_2 \land asc-fault-domain \cdot M_2 \cdot M_1 \ m \land test-tools \cdot M_2 \cdot M_1 \ FAIL \ PM \ V \ \Omega\]
\[\}\]
DO
\[iter := iter + 1;\]
\[rmN := \{xs' \in cN .\]
\[\neg (L_n, M_1 \cdot \{xs'\} \subseteq L_n \cdot M_2 \cdot \{xs'\}))\]
\[\lor (\forall \text{io} \in L_n, M_1 \cdot \{xs'\} .\]
\[\exists V'' \in N \cdot io \cdot M_1 \ V .\]
\[\exists S1 .\]
\[\exists vs \ xs .\]
\[\text{io} = (vs@xs)\]
\[\land \ mcp \ (vs@xs) \ V'' \ vs\]
\[\land S1 \subseteq \ nodes \ M_2\]
\[\land (\forall s1 \in S1 . \forall s2 \in S1 .\]
\[s1 \neq s2 \rightarrow\]
\[\forall \text{io}1 \in RP \cdot M_2 \ s1 \ vs \ xs \ V'' .\]
\[\forall \text{io}2 \in RP \cdot M_2 \ s2 \ vs \ xs \ V'' .\]
\[B \ M_1 \text{io} \ \Omega \neq B \ M_1 \text{io}2 \ \Omega)\]
\[\land m < LB \ M_2 \ M_1 \ vs \ xs \ (tsN \cup V) \ S1 \ \Omega \ V'' ))));\]
\[tsN := tsN \cup cN;\]
\[cN := append-set \ (cN - rmN) \ (inputs \ M_2) - tsN;\]
\[obs := obs \cup L_n \cdot M_2 \ cN;\]
\[obsI := obsI \cup L_n \cdot M_1 \ cN;\]
\[obs \Omega := obs \Omega \cup (\bigcup \text{io} \in L_n, M_2 \cdot cN. \{\text{io}\} \times B \cdot M_2 \cdot \text{io} \cdot \Omega);\]
\[obsI \Omega := obsI \Omega \cup (\bigcup \text{io} \in L_n, M_1 \cdot cN. \{\text{io}\} \times B \cdot M_1 \cdot \text{io} \cdot \Omega);\]
OD;
\[\text{isReduction} := ((obsI \subseteq obs) \land (obsI \Omega \subseteq obsI \Omega))\]
\[\{\]
\[\text{isReduction} = M_1 \leq M_2 \ — \ variable \ isReduction \ is \ used \ only \ as \ a \ return \ value, \ it \ is \ true \ if \ and \ only \ if \ M_1 \ is \ a \ reduction \ of \ M_2\]
\[\}\]
(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.

```haskell
fun from-rel :: ('state × ('in × 'out) × 'state) set ⇒ 'state ⇒ ('in, 'out, 'state) FSM where
from-rel q0 = ⌾ succ = λ p q. \( p, (p,io,q) ∈ rel \),
inputs = image (fst o fst o snd) rel,
outputs = image (snd o fst o snd) rel,
initial = q0 ⌾
```

```haskell
lemma nodes-from-rel : nodes (from-rel q0) ⊆ insert q0 (image (snd o snd) rel)
(is nodes ?M ⊆ insert q0 (image (snd o snd) rel))
(proof)
```

```haskell
fun well-formed-rel :: ('state × ('in × 'out) × 'state) set ⇒ bool where
well-formed-rel rel = (finite rel ∧ (∀ s1 x y. (x \∉ image (fst o fst o snd) rel) ∨ y \∉ image (snd o fst o snd) rel) → ¬(∃ s2 . (s1,(x,y),s2) ∈ rel)) ∧ rel \neq \{\}
```

```haskell
lemma well-formed-from-rel :
assumes well-formed-rel rel
shows well-formed (from-rel rel q0) (is well-formed ?M)
(proof)
```

```haskell
fun completely-specified-rel-over :: ('state × ('in × 'out) × 'state) set ⇒ bool where
completely-specified-rel-over rel nods = (∀ s1 \in nods .
∀ x \in image (fst o fst o snd) rel .
∃ y \in image (snd o fst o snd) rel .
∃ s2 . (s1,(x,y),s2) ∈ 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)
(proof)
```

```haskell
fun observable-rel :: ('state × ('in × 'out) × 'state) set ⇒ bool where
observable-rel rel = (∀ io s1 . \{ s2 . (s1,io,s2) ∈ rel \} = \{\}) ∨ (∃ s2 . \{ s2' . (s1,io,s2') ∈ rel \} = \{s2\})
lemma observable-from-rel :
assumes observable-rel rel
shows observable (from-rel rel q0) (is observable ?M)
(proof)
```

```haskell
abbreviation OFSM-rel rel q0 ≡ well-formed-rel rel
∧ completely-specified-rel-over rel (nodes (from-rel rel q0))
∧ observable-rel rel
```
lemma OFMS-from-rel:
  assumes OFSM-rel rel q0
  shows OFSM (from-rel rel q0)
  ⟨proof⟩

8.2 Example FSMs and properties

abbreviation M_S-rel :: (nat×(nat×nat)) set ≡ \{(0,(0,0),1), (0,(0,1),1), (1,(0,0),0)\}
abbreviation M_I-rel :: (nat×(nat×nat)) set ≡ \{(0,(0,0),1), (0,(0,1),1), (1,(0,2),0)\}

lemma example-nodes:
  nodes M_S = {0,1} nodes M_I = {0,1}
  ⟨proof⟩

abbreviation M_S :: (nat, nat, nat) FSM ≡ from-rel M_S-rel 0
abbreviation M_I :: (nat, nat, nat) FSM ≡ from-rel M_I-rel 0

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

abbreviation M_S-rel I :: (nat×nat) set ≡ \{(0,0), (0,1), (1,1)\}
abbreviation M_I-rel I :: (nat×nat) set ≡ \{(0,0), (0,1), (1,2)\}

abbreviation example-fault-domain:
  asc-fault-domain M_S M_I
  ⟨proof⟩

abbreviation FAIL_I :: (nat×nat) ≡ (3,3)
abbreviation PM_I :: (nat, nat, nat) FSM ≡ \|
  succ = (\lambda a (p1,p2) . (if (p1 ∈ nodes M_S ∧ p2 ∈ nodes M_I) ∧ (fst a ∈ inputs M_S)
    ∧ (snd a ∈ outputs M_S ∪ outputs M_I))
    then (if (succ M_S a p1 = {} ∧ succ M_I a p2 = {}) then {FAIL_I} else (succ M_S a p1 × succ M_I a p2))
    else {})),
  inputs = inputs M_S,
  outputs = outputs M_S ∪ 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 ≡ \{[],[0]\}
lemma example-det-state-cover:
  is-det-state-cover M_S V_I
  ⟨proof⟩

abbreviation Ω_I::(nat,nat) ATC set ≡ \{ Node 0 (\lambda y . Leaf) \}
lemma applicable-set M_S Ω_I
  ⟨proof⟩

abbreviation example-test-tools:
  test-tools M_S M_I FAIL_I PM_I V_I Ω_I
  ⟨proof⟩

lemma OFSM-not-vacuous:
\[ \exists M :: (\text{nat, nat, nat}) \text{ FSM} \ . \ \text{OFSM} M \]

**Lemma** fault-domain-not-vacuous :
\[ \exists (M2::(\text{nat, nat, nat}) \text{ FSM}) (M1::(\text{nat, nat, nat}) \text{ FSM}) \ m \ . \ \text{asc-fault-domain} M2 M1 m \]

**Lemma** test-tools-not-vacuous :
\[ \exists (M2::(\text{nat, nat, nat}) \text{ FSM})  \\ (M1::(\text{nat, nat, nat}) \text{ FSM})  \\ (\text{FAIL}::(\text{nat} \times \text{nat}))  \\ (PM::(\text{nat, nat, nat} \times \text{nat}) \text{ FSM})  \\ (V::(\text{nat list set}))  \\ (\Omega::(\text{nat, nat}) \text{ ATC set}) \ . \ \text{test-tools} M2 M1 \text{ FAIL} PM V \Omega \]

**Lemma** precondition-not-vacuous :
\[ \exists (M2::(\text{nat, nat, nat}) \text{ FSM})  \\ (M1::(\text{nat, nat, nat}) \text{ FSM})  \\ (\text{FAIL}::(\text{nat} \times \text{nat}))  \\ (PM::(\text{nat, nat, nat} \times \text{nat}) \text{ FSM})  \\ (V::(\text{nat list set}))  \\ (\Omega::(\text{nat, nat}) \text{ ATC set})  \\ (m::\text{nat}) \ . \ \text{OFSM} M1 \land \text{OFSM} M2 \land \text{asc-fault-domain} M2 M1 m \land \text{test-tools} M2 M1 \text{ FAIL} PM V \Omega \]

**References**

