

Formalization of Dynamic Pushdown Networks in Isabelle/HOL

Peter Lammich

January 26, 2026

Abstract

We present a formalization of Dynamic Pushdown Networks (DPNs) and the automata based algorithm for computing backward reachability sets using Isabelle/HOL. Dynamic pushdown networks are an abstract model for multithreaded, interprocedural programs with dynamic thread creation that was presented by Bouajjani, Müller-Olm and Touili in 2005.

We formalize the notion of a DPN in Isabelle and describe the algorithm for computing the pre^* -set from a regular set of configurations, and prove its correctness. We first give a nondeterministic description of the algorithm, from that we then infer a deterministic one, from which we can generate executable code using Isabelle's code-generation tool.

Contents

1	String rewrite systems	3
1.1	Definitions	3
1.2	Induced Labelled Transition System	3
1.3	Properties of the induced LTS	3
2	Finite state machines	4
2.1	Definitions	4
2.2	Basic properties	5
2.3	Constructing FSMs	5
2.4	Reflexive, transitive closure of transition relation	6
2.4.1	Relation of $trclAD$ and $trcl$	7
2.5	Language of a FSM	8
2.6	Example: Product automaton	8

3 Nondeterministic recursive algorithms	9
3.1 Basic properties	10
3.2 Refinement	11
3.3 Extension to reflexive states	13
3.4 Well-foundedness	14
3.4.1 The relations $>$ and \supset on finite domains	15
3.5 Implementation	16
3.5.1 Graphs of functions	17
3.5.2 Deterministic refinement w.r.t. the identity abstraction	17
3.5.3 Recursive characterization	17
4 Dynamic pushdown networks	18
4.1 Dynamic pushdown networks	19
4.1.1 Definition	19
4.1.2 Basic properties	19
4.1.3 Building DPNs	20
4.2 M-automata	22
4.2.1 Definition	22
4.2.2 Basic properties	23
4.2.3 Some implications of the M-automata conditions	24
4.3 pre^* -sets of regular sets of configurations	26
4.4 Nondeterministic algorithm for pre^*	27
4.4.1 Termination	28
4.4.2 Soundness	31
4.4.3 Precision	33
5 Non-executable implementation of the DPN pre^*-algorithm	40
5.1 Definitions	40
5.2 Refining $ps\text{-}R$	42
5.3 Termination	42
5.4 Recursive characterization	43
5.5 Correctness	44
6 Tools for executable specifications	44
6.1 Searching in Lists	44
7 Executable algorithms for finite state machines	45
7.1 Word lookup operation	45
7.2 Reachable states and alphabet inferred from transition relation	47
8 Implementation of DPN pre^*-algorithm	48
8.1 Representation of DPN and M-automata	48
8.2 Next-element selection	49
8.3 Termination	50

8.3.1	Saturation upper bound	50
8.3.2	Well-foundedness of recursion relation	52
8.3.3	Definition of recursive function	52
8.4	Correctness	53
8.4.1	seln_R refines ps_R	53
8.4.2	Computing transitions only	56
8.4.3	Correctness	56

1 String rewrite systems

```
theory SRS
imports DPN-Setup
begin
```

This formalizes systems of labelled string rewrite rules and the labelled transition systems induced by them. DPNs are special string rewrite systems.

1.1 Definitions

```
type-synonym ('c,'l) rewrite-rule = 'c list × 'l × 'c list
type-synonym ('c,'l) SRS = ('c,'l) rewrite-rule set
```

syntax

```
syn-rew-rule :: 'c list ⇒ 'l ⇒ 'c list ⇒ ('c,'l) rewrite-rule (- ↪ - [51,51,51] 51)
```

translations

```
 $s \xrightarrow{a} s' \Rightarrow (s, a, s')$ 
```

A (labelled) rewrite rule (s, a, s') consists of the left side s , the label a and the right side s' . Intuitively, it means that a substring s can be rewritten to s' by an a -step. A string rewrite system is a set of labelled rewrite rules

1.2 Induced Labelled Transition System

A string rewrite systems induces a labelled transition system on strings by rewriting substrings according to the rules

```
inductive-set tr :: ('c,'l) SRS ⇒ ('c list, 'l) LTS for S
```

where

```
rewrite:  $(s \xrightarrow{a} s') \in S \implies (ep@s@es, a, ep@s'@es) \in tr S$ 
```

1.3 Properties of the induced LTS

Adding characters at the start or end of a state does not influence the capability of making a transition

```
lemma srs-ext-s:  $(s, a, s') \in tr S \implies (wp@s@ws, a, wp@s'@ws) \in tr S$  proof –
assume  $(s, a, s') \in tr S$ 
```

```

then obtain  $ep \ es \ r \ r'$  where  $s = ep @ r @ es \wedge s' = ep @ r' @ es \wedge (r, a, r') \in S$  by (fast
elim: tr.cases)
moreover hence  $((wp @ ep) @ r @ (es @ ws), a, (wp @ ep) @ r' @ (es @ ws)) \in tr S$  by (fast
intro: tr.rewrite)
ultimately show ?thesis by auto
qed

lemma srs-ext-both:  $(s, w, s') \in trcl (tr S) \implies (wp @ s @ ws, w, wp @ s' @ ws) \in trcl (tr S)$ 
apply (induct s w s' rule: trcl.induct)
apply (simp)
apply (subgoal-tac wp @ c @ ws  $\hookrightarrow_a wp @ c' @ ws \in tr S)
apply (auto intro: srs-ext-s)
done

corollary srs-ext-cons:  $(s, w, s') \in trcl (tr S) \implies (e \# s, w, e \# s') \in trcl (tr S)$  by (rule
srs-ext-both[where wp=[e] and ws=[], simplified])
corollary srs-ext-pre:  $(s, w, s') \in trcl (tr S) \implies (wp @ s, w, wp @ s') \in trcl (tr S)$  by (rule
srs-ext-both[where ws=[], simplified])
corollary srs-ext-post:  $(s, w, s') \in trcl (tr S) \implies (s @ ws, w, s' @ ws) \in trcl (tr S)$  by (rule
srs-ext-both[where wp=[], simplified])

lemmas srs-ext = srs-ext-both srs-ext-pre srs-ext-post$ 
```

```
end
```

2 Finite state machines

```

theory FSM
imports DPN-Setup
begin

```

This theory models nondeterministic finite state machines with explicit set of states and alphabet. ε -transitions are not supported.

2.1 Definitions

```

record ('s,'a) FSM-rec =
   $Q :: 's \text{ set}$  — The set of states
   $\Sigma :: 'a \text{ set}$  — The alphabet
   $\delta :: ('s, 'a) \text{ LTS}$  — The transition relation
   $s0 :: 's$  — The initial state
   $F :: 's \text{ set}$  — The set of final states

```

```

locale FSM =
  fixes A
  assumes delta-cons:  $(q, l, q') \in \delta \ A \implies q \in Q \ A \wedge l \in \Sigma \ A \wedge q' \in Q \ A$  — The transition
  relation is consistent with the set of states and the alphabet
  assumes s0-cons:  $s0 \ A \in Q \ A$  — The initial state is a state

```

assumes $F\text{-cons}$: $F A \subseteq Q A$ — The final states are states
assumes finite-states : $\text{finite}(Q A)$ — The set of states is finite
assumes finite-alphabet : $\text{finite}(\Sigma A)$ — The alphabet is finite

2.2 Basic properties

lemma (in FSM) finite-delta-dom : $\text{finite}(Q A \times \Sigma A \times Q A)$ **proof** —
from finite-states finite-alphabet $\text{finite-cartesian-product}[\text{of } \Sigma A \text{ } Q A]$ **have** $\text{finite}(\Sigma A \times Q A)$ **by** *fast*
with finite-states $\text{finite-cartesian-product}[\text{of } Q A \Sigma A \times Q A]$ **show** $\text{finite}(Q A \times \Sigma A \times Q A)$ **by** *fast*
qed

lemma (in FSM) finite-delta : $\text{finite}(\delta A)$ **proof** —
have $\delta A \subseteq Q A \times \Sigma A \times Q A$ **by** *(auto simp add: delta-cons)*
with finite-delta-dom **show** $?thesis$ **by** *(simp add: finite-subset)*
qed

2.3 Constructing FSMs

definition $fsm-empty$ $s_0 \equiv () \ Q=\{s_0\}, \Sigma=\{\}, \delta=\{\}, s0=s_0, F=\{\} ()$
definition $fsm-add-F$ $s fsm \equiv fsm() \ Q:=\text{insert } s (Q fsm), F:=\text{insert } s (F fsm) ()$
definition $fsm-add-tr$ $q \ a \ q' fsm \equiv fsm() \ Q:=\{q, q'\} \cup (Q fsm), \Sigma:=\text{insert } a (\Sigma fsm), \delta := \text{insert } (q, a, q') (\delta fsm) ()$

lemma $fsm-empty-invar$ [*simp*]: $\text{FSM}(fsm-empty s)$
apply *unfold-locales* **unfolding** $fsm-empty$ -def **by** *auto*

lemma $fsm-add-F-invar$ [*simp*]: **assumes** $\text{FSM } fsm$ **shows** $\text{FSM}(fsm-add-F s fsm)$

proof —
interpret $\text{FSM } fsm$ **by** *fact*
show $?thesis$
apply *unfold-locales*
unfolding $fsm-add-F$ -def
using delta-cons $s0\text{-cons}$ $F\text{-cons}$ finite-states finite-alphabet
by *auto*
qed

lemma $fsm-add-tr-invar$ [*simp*]: **assumes** $\text{FSM } fsm$ **shows** $\text{FSM}(fsm-add-tr q a q' fsm)$
proof —
interpret $\text{FSM } fsm$ **by** *fact*
show $?thesis$
apply *unfold-locales*
unfolding $fsm-add-tr$ -def
using delta-cons $s0\text{-cons}$ $F\text{-cons}$ finite-states finite-alphabet
by *auto*
qed

2.4 Reflexive, transitive closure of transition relation

Reflexive transitive closure on restricted domain

```
inductive-set trclAD :: ('s,'a,'c) FSM-rec-scheme ⇒ ('s,'a) LTS ⇒ ('s,'a list)
LTS
for A D
where
empty[simp]: s ∈ Q A ⇒ (s, [], s) ∈ trclAD A D |
cons[simp]: [(s, e, s') ∈ D; s ∈ Q A; e ∈ Σ A; (s', w, s'') ∈ trclAD A D] ⇒ (s, e # w, s'') ∈ trclAD A D
```

abbreviation trclA A == trclAD A (δ A)

```
lemma trclAD-empty-cons[simp]: (c, [], c') ∈ trclAD A D ⇒ c = c' by (auto elim: trclAD.cases)
lemma trclAD-single: (c, [a], c') ∈ trclAD A D ⇒ (c, a, c') ∈ D by (auto elim: trclAD.cases)
lemma trclAD-elems: (c, w, c') ∈ trclAD A D ⇒ c ∈ Q A ∧ w ∈ lists (Σ A) ∧ c' ∈ Q A by (erule trclAD.induct, auto)
lemma trclAD-one-elem: [(c ∈ Q A; e ∈ Σ A; c' ∈ Q A; (c, e, c') ∈ D)] ⇒ (c, [e], c') ∈ trclAD A D by auto

lemma trclAD-uncons: (c, a # w, c') ∈ trclAD A D ⇒ ∃ ch . (c, a, ch) ∈ D ∧ (ch, w, c') ∈ trclAD A D ∧ c ∈ Q A ∧ a ∈ Σ A
by (auto elim: trclAD.cases)

lemma trclAD-concat: !! c . [(c, w1, c') ∈ trclAD A D; (c', w2, c'') ∈ trclAD A D] ⇒ (c, w1 @ w2, c'') ∈ trclAD A D
proof (induct w1)
  case Nil thus ?case by (subgoal-tac c = c') auto
next
  case (Cons a w) thus ?case by (auto dest: trclAD-uncons)
qed

lemma trclAD-unconcat: !! c . (c, w1 @ w2, c') ∈ trclAD A D ⇒ ∃ ch . (c, w1, ch) ∈ trclAD A D ∧ (ch, w2, c') ∈ trclAD A D proof (induct w1)
  case Nil hence (c, [], c) ∈ trclAD A D ∧ (c, w2, c') ∈ trclAD A D by (auto dest: trclAD-elems)
  thus ?case by fast
next
  case (Cons a w1) note IHP = this
  hence (c, a # (w1 @ w2), c') ∈ trclAD A D by simp
  with trclAD-uncons obtain chh where (c, a, chh) ∈ D ∧ (chh, w1 @ w2, c') ∈ trclAD A D ∧ c ∈ Q A ∧ a ∈ Σ A by fast
  moreover with IHP obtain ch where (chh, w1, ch) ∈ trclAD A D ∧ (ch, w2, c') ∈ trclAD A D by fast
  ultimately have (c, a # w1, ch) ∈ trclAD A D ∧ (ch, w2, c') ∈ trclAD A D by auto
```

```
thus ?case by fast
qed
```

```
lemma trclAD-eq:  $\llbracket Q A = Q A'; \Sigma A = \Sigma A' \rrbracket \implies \text{trclAD } A D = \text{trclAD } A' D$ 
  apply (safe)
  subgoal by (erule trclAD.induct) auto
  subgoal by (erule trclAD.induct) auto
  done
```

```
lemma trclAD-mono:  $D \subseteq D' \implies \text{trclAD } A D \subseteq \text{trclAD } A D'$ 
  apply (clarify)
  apply (erule trclAD.induct)
  apply auto
  done
```

```
lemma trclAD-mono-adv:  $\llbracket D \subseteq D'; Q A = Q A'; \Sigma A = \Sigma A' \rrbracket \implies \text{trclAD } A D \subseteq \text{trclAD } A' D'$  by (subgoal-tac trclAD A D = trclAD A' D) (auto dest: trclAD-eq trclAD-mono)
```

2.4.1 Relation of trclAD and trcl

```
lemma trclAD-by-trcl1:  $\text{trclAD } A D \subseteq (\text{trcl } (D \cap (Q A \times \Sigma A \times Q A)) \cap (Q A \times \text{lists } (\Sigma A) \times Q A))$ 
  by (auto 0 3 dest: trclAD-elems elim: trclAD.induct simp: trclAD-elems intro: trcl.cons)
```

```
lemma trclAD-by-trcl2:  $(\text{trcl } (D \cap (Q A \times \Sigma A \times Q A)) \cap (Q A \times \text{lists } (\Sigma A) \times Q A)) \subseteq \text{trclAD } A D$  proof -
  { fix c
    have !! s s'.  $\llbracket (s, c, s') \in \text{trcl } (D \cap Q A \times \Sigma A \times Q A); s \in Q A; s' \in Q A; c \in \text{lists } (\Sigma A) \rrbracket \implies (s, c, s') \in \text{trclAD } A D$  proof (induct c)
      case Nil thus ?case by (auto dest: trcl-empty-cons)
      next
        case (Cons e w) note IHP=this
        then obtain sh where SPLIT:  $(s, e, sh) \in (D \cap Q A \times \Sigma A \times Q A) \wedge (sh, w, s') \in \text{trcl } (D \cap Q A \times \Sigma A \times Q A)$  by (fast dest: trcl-uncons)
        hence  $(sh, w, s') \in \text{trcl } (D \cap Q A \times \Sigma A \times Q A) \cap (Q A \times \text{lists } (\Sigma A) \times Q A)$  by (auto elim!: trcl-structE)
        hence  $(sh, w, s') \in \text{trclAD } A D$  by (blast intro: IHP)
        with SPLIT show ?case by auto
      qed
    }
    thus ?thesis by (auto)
  qed
```

```
lemma trclAD-by-trcl:  $\text{trclAD } A D = (\text{trcl } (D \cap (Q A \times \Sigma A \times Q A)) \cap (Q A \times \text{lists } (\Sigma A) \times Q A))$ 
  apply (rule equalityI)
  apply (rule trclAD-by-trcl1)
```

```

apply (rule trclAD-by-trcl2)
done

lemma trclAD-by-trcl': trclAD A D = (trcl (D ∩ (Q A × Σ A × Q A)) ∩ (Q A × UNIV × UNIV))
by (auto iff add: trclAD-by-trcl elim!: trcl-structE)

lemma trclAD-by-trcl'':  $\llbracket D \subseteq Q A \times \Sigma A \times Q A \rrbracket \implies \text{trclAD } A D = \text{trcl } D \cap (Q A \times \Sigma A \times Q A)$ 
using trclAD-by-trcl'[of A D] by (simp add: Int-absorb2)

lemma trclAD-subset-trcl: trclAD A D  $\subseteq \text{trcl } (D) proof –
  have trclAD A D  $\subseteq (\text{trcl } (D \cap (Q A \times \Sigma A \times Q A)))$  by (auto simp add: trclAD-by-trcl)
  also with trcl-mono[of D ∩ (Q A × Σ A × Q A) D] have  $\dots \subseteq \text{trcl } D$  by auto
  finally show ?thesis .
qed$ 
```

2.5 Language of a FSM

```

definition langs A s == { w .  $(\exists f \in (F A) . (s, w, f) \in \text{trclA } A)$  }
definition lang A == langs A (s0 A)

```

```

lemma langs-alt-def:  $(w \in \text{langs } A s) \iff (\exists f . f \in F A \wedge (s, w, f) \in \text{trclA } A)$  by (intro eq-reflection, unfold langs-def, auto)

```

2.6 Example: Product automaton

```

definition prod-fsm A1 A2 ==  $\langle Q = Q A1 \times Q A2, \Sigma = \Sigma A1 \cap \Sigma A2, \delta = \{ ((s, t), a, (s', t')) . (s, a, s') \in \delta A1 \wedge (t, a, t') \in \delta A2 \}, s0 = (s0 A1, s0 A2), F = \{(s, t) . s \in F A1 \wedge t \in F A2\} \rangle$ 

```

```

lemma prod-inter-1:  $\exists s s' f f' . ((s, s'), w, (f, f')) \in \text{trclA } (\text{prod-fsm } A A') \implies (s, w, f) \in \text{trclA } A \wedge (s', w, f') \in \text{trclA } A'$  proof (induct w)
  case Nil note P=this
  moreover hence s=f  $\wedge$  s'=f' by (fast dest: trclAD-empty-cons)
  moreover from P have s ∈ Q A  $\wedge$  s' ∈ Q A' by (unfold prod-fsm-def, auto dest: trclAD-elems)
  ultimately show ?case by (auto)
next
  case (Cons e w)
  note IHP=this
  then obtain m m' where I:  $((s, s'), e, (m, m')) \in \delta (\text{prod-fsm } A A') \wedge (s, s') \in Q (\text{prod-fsm } A A') \wedge e \in \Sigma (\text{prod-fsm } A A') \wedge ((m, m'), w, (f, f')) \in \text{trclA } (\text{prod-fsm } A A')$  by (fast dest: trclAD-uncons)
  hence  $(s, e, m) \in \delta A \wedge (s', e, m') \in \delta A' \wedge s \in Q A \wedge s' \in Q A' \wedge e \in \Sigma A \wedge e \in \Sigma A'$  by (unfold prod-fsm-def, simp)
  moreover from I IHP have  $(m, w, f) \in \text{trclA } A \wedge (m', w, f') \in \text{trclA } A'$  by auto
  ultimately show ?case by auto
qed

```

```

lemma prod-inter-2: !! s s' f f' . (s,w,f) ∈ trclA A ∧ (s',w,f') ∈ trclA A' ⇒
((s,s'),w,(f,f')) ∈ trclA (prod-fsm A A') proof (induct w)
  case Nil note P=this
  moreover hence s=f ∧ s'=f' by (fast dest: trclAD-empty-cons)
  moreover from P have (s,s') ∈ Q (prod-fsm A A') by (unfold prod-fsm-def, auto
dest: trclAD-elems)
  ultimately show ?case by simp
next
  case (Cons e w)
  note IHP=this
  then obtain m m' where I: (s,e,m) ∈ δ A ∧ (m,w,f) ∈ trclA A ∧ (s',e,m') ∈ δ
A' ∧ (m',w,f') ∈ trclA A' ∧ s ∈ Q A ∧ s' ∈ Q A' ∧ e ∈ Σ A ∧ e ∈ Σ A' by (fast dest:
trclAD-uncons)
  hence ((s,s'),e,(m,m')) ∈ δ (prod-fsm A A') ∧ (s,s') ∈ Q (prod-fsm A A') ∧ e ∈ Σ
(prod-fsm A A') by (unfold prod-fsm-def, simp)
  moreover from I IHP have ((m,m'),w,(f,f')) ∈ trclA (prod-fsm A A') by auto
  ultimately show ?case by auto
qed

lemma prod-F: (a,b) ∈ F (prod-fsm A B) = (a ∈ F A ∧ b ∈ F B) by (unfold prod-fsm-def,
auto)
lemma prod-FI: [a ∈ F A; b ∈ F B] ⇒ (a,b) ∈ F (prod-fsm A B) by (unfold prod-fsm-def,
auto)

lemma prod-fsm-langs: langs (prod-fsm A B) (s,t) = langs A s ∩ langs B t
  apply (unfold langs-def)
  apply (insert prod-inter-1 prod-F)
  apply (fast intro: prod-inter-2 prod-FI)
done

lemma prod-FSM-intro: FSM A1 ⇒ FSM A2 ⇒ FSM (prod-fsm A1 A2) by
(rule FSM.intro) (auto simp add: FSM-def prod-fsm-def)

end

```

3 Nondeterministic recursive algorithms

```

theory NDET
imports Main
begin

```

This theory models nondeterministic, recursive algorithms by means of a step relation.

An algorithm is modelled as follows:

1. Start with some state s

2. If there is no s' with $(s, s') \in R$, terminate with state s
3. Else set $s := s'$ and continue with step 2

Thus, R is the step relation, relating the previous with the next state. If the state is not in the domain of R , the algorithm terminates.

The relation $A\text{-rel } R$ describes the non-reflexive part of the algorithm, that is all possible mappings for non-terminating initial states. We will first explore properties of this non-reflexive part, and then transfer them to the whole algorithm, that also specifies how terminating initial states are treated.

```
inductive-set A-rel :: ('s×'s) set ⇒ ('s×'s) set for R
where
  A-rel-base: [(s,s') ∈ R; s' ∉ Domain R] ⇒ (s,s') ∈ A-rel R |
  A-rel-step: [(s,sh) ∈ R; (sh,s') ∈ A-rel R] ⇒ (s,s') ∈ A-rel R
```

3.1 Basic properties

The algorithm just terminates at terminating states

```
lemma termstate: (s,s') ∈ A-rel R ⇒ s' ∉ Domain R by (induct rule: A-rel.induct, auto)
```

```
lemma dom-subset: Domain (A-rel R) ⊆ Domain R by (unfold Domain-def) (auto elim: A-rel.induct)
```

We can use invariants to reason over properties of the algorithm

```
definition is-inv R s0 P == P s0 ∧ (∀ s s'. (s,s') ∈ R ∧ P s → P s')
```

```
lemma inv: [(s0,sf) ∈ A-rel R; is-inv R s0 P] ⇒ P sf by (unfold is-inv-def, induct rule: A-rel.induct) blast+
lemma invI: [P s0; !! s s'. [(s,s') ∈ R; P s] ⇒ P s'] ⇒ is-inv R s0 P by (unfold is-inv-def, blast)
lemma inv2: [(s0,sf) ∈ A-rel R; P s0; !! s s'. [(s,s') ∈ R; P s] ⇒ P s'] ⇒ P sf
  apply (subgoal-tac is-inv R s0 P)
  apply (blast intro: inv)
  apply (blast intro: invI)
done
```

To establish new invariants, we can use already existing invariants

```
lemma inv-useI: [P s0; !! s s'. [(s,s') ∈ R; P s; !!P'. is-inv R s0 P' ⇒ P' s] ⇒ P s'] ⇒ is-inv R s0 (λs. P s ∧ (∀ P'. is-inv R s0 P' → P' s))
  apply (rule invI)
  apply (simp (no-asm) only: is-inv-def, blast)
  apply safe
  apply blast
  apply (subgoal-tac P' s)
  apply (simp (no-asm-use) only: is-inv-def, blast)
  apply fast
```

done

If the inverse step relation is well-founded, the algorithm will terminate for every state in *Domain R* (\subseteq -direction). The \supseteq -direction is from *dom-subset*

```

lemma wf-dom-eq: wf (R-1)  $\implies$  Domain R = Domain (A-rel R) proof -
  assume WF: wf (R-1)
  hence ( $\exists$  sf. (s,sf)  $\in$  A-rel R) if (s,s')  $\in$  R for s s' using that
  proof (induction arbitrary: s')
    case (less x)

  {
    assume s'  $\notin$  Domain R
    with less.prem have (x,s')  $\in$  A-rel R by (blast intro: A-rel-base)
  } moreover {
    assume s'  $\in$  Domain R
    then obtain st where (s',st)  $\in$  R by (unfold Domain-def, auto)
    with less.prem less.IH obtain sf where (s',sf)  $\in$  A-rel R by blast
    with less.prem have (x,sf)  $\in$  A-rel R by (blast intro: A-rel-step)
    hence  $\exists$  sf. (x,sf)  $\in$  A-rel R by blast
  } ultimately show  $\exists$  sf. (x,sf)  $\in$  A-rel R by blast
qed
hence Domain R  $\subseteq$  Domain (A-rel R) by (unfold Domain-def, auto)
with dom-subset show ?thesis by force
qed

```

3.2 Refinement

Refinement is a simulation property between step relations.

We define refinement w.r.t. an abstraction relation α , that relates abstract to concrete states. The refining step-relation is called more concrete than the refined one.

```

definition refines :: ('s*'s) set  $\Rightarrow$  ('r*'s) set  $\Rightarrow$  ('r*'r) set  $\Rightarrow$  bool (- $\leq$ - [50,50,50]
50) where
  R  $\leq_\alpha$  S ==  $\alpha$  O R  $\subseteq$  S O  $\alpha$   $\wedge$   $\alpha$  “ Domain S  $\subseteq$  Domain R

lemma refinesI:  $\llbracket \alpha$  O R  $\subseteq$  S O  $\alpha$ ;  $\alpha$  “ Domain S  $\subseteq$  Domain R  $\rrbracket \implies$  R  $\leq_\alpha$  S by
  (unfold refines-def, auto)
lemma refinesE: R  $\leq_\alpha$  S  $\implies$   $\alpha$  O R  $\subseteq$  S O  $\alpha$ 
  R  $\leq_\alpha$  S  $\implies$   $\alpha$  “ Domain S  $\subseteq$  Domain R
  by (unfold refines-def, auto)

```

Intuitively, the first condition for refinement means, that for each concrete step $(c,c') \in S$ where the start state c has an abstract counterpart $(a,c) \in \alpha$, there is also an abstract counterpart of the end state $(a',c') \in \alpha$ and the step can also be done on the abstract counterparts $(a,a') \in R$.

```

lemma refines-compI:
  assumes A: !! a c c'.  $\llbracket (a,c) \in \alpha; (c,c') \in S \rrbracket \implies \exists a'. (a,a') \in R \wedge (a',c') \in \alpha$ 

```

shows $\alpha \ O \ S \subseteq R \ O \ \alpha$ **using** A **by** *blast*

lemma *refines-compE*: $\llbracket \alpha \ O \ S \subseteq R \ O \ \alpha; (a,c) \in \alpha; (c,c') \in S \rrbracket \implies \exists a'. (a,a') \in R \wedge (a',c') \in \alpha$ **by** *(auto)*

Intuitively, the second condition for refinement means, that if there is an abstract step $(a,a') \in R$, where the start state has a concrete counterpart c , then there must also be a concrete step from c . Note that this concrete step is not required to lead to the concrete counterpart of a' . In fact, it is only important that there is such a concrete step, ensuring that the concrete algorithm will not terminate on states on that the abstract algorithm continues execution.

lemma *refines-domI*:

assumes $A: !! a \ a' \ c. \llbracket (a,c) \in \alpha; (a,a') \in R \rrbracket \implies c \in \text{Domain } S$
shows $\alpha \text{ `` Domain } R \subseteq \text{Domain } S$ **using** A **by** *auto*

lemma *refines-domE*: $\llbracket \alpha \text{ `` Domain } R \subseteq \text{Domain } S; (a,c) \in \alpha; (a,a') \in R \rrbracket \implies c \in \text{Domain } S$ **by** *auto*

lemma *refinesI2*:

assumes $A: !! a \ c \ c'. \llbracket (a,c) \in \alpha; (c,c') \in S \rrbracket \implies \exists a'. (a,a') \in R \wedge (a',c') \in \alpha$

assumes $B: !! a \ a' \ c. \llbracket (a,c) \in \alpha; (a,a') \in R \rrbracket \implies c \in \text{Domain } S$

shows $S \leq_{\alpha} R$ **by** *(simp only: refinesI A refines-compI B refines-domI)*

lemma *refinesE2*:

$\llbracket S \leq_{\alpha} R; (a,c) \in \alpha; (c,c') \in S \rrbracket \implies \exists a'. (a,a') \in R \wedge (a',c') \in \alpha$

$\llbracket S \leq_{\alpha} R; (a,c) \in \alpha; (a,a') \in R \rrbracket \implies c \in \text{Domain } S$

by *(blast dest: refinesE refines-compE refines-domE) +*

Reflexivity of identity refinement

lemma *refines-id-refl[intro!, simp]*: $R \leq_{Id} R$ **by** *(auto intro: refinesI)*

Transitivity of refinement

lemma *refines-trans*: **assumes** $R: R \leq_{\alpha} S \quad S \leq_{\beta} T$ **shows** $R \leq_{\beta} O \alpha T$
proof *(rule refinesI)*

{

fix $s \ s' \ t'$

assume $A: (s,s') \in \beta \ O \ \alpha \ (s',t') \in R$

then obtain sh **where** $(s,sh) \in \beta \wedge (sh,s') \in \alpha$ **by** *(blast)*

with $A \ R$ **obtain** $t \ th$ **where** $(sh,th) \in S \wedge (th,t') \in \alpha \wedge (s,t) \in T \wedge (t,th) \in \beta$ **by**

(blast dest: refinesE)

hence $(s,t') \in T \ O \ (\beta \ O \ \alpha)$ **by** *blast*

} thus $(\beta \ O \ \alpha) \ O \ R \subseteq T \ O \ (\beta \ O \ \alpha)$ **by** *blast*

next

{

fix $s \ s'$

assume $A: s \in \text{Domain } T \ (s,s') \in \beta \ O \ \alpha$

then obtain sh **where** $(s,sh) \in \beta \wedge (sh,s') \in \alpha$ **by** *blast*

with R A **have** $s' \in \text{Domain } R$ **by** (blast dest!: refinesE)
} thus $(\beta O \alpha)$ “ Domain $T \subseteq \text{Domain } R$ **by** (unfold Domain-def, blast)
qed

Property transfer lemma

lemma refines-A-rel[rule-format]:
assumes $R: R \leq_{\alpha} S$ **and** $A: (r, r') \in A\text{-rel } R$ $(s, r) \in \alpha$
shows $(\exists s'. (s', r') \in \alpha \wedge (s, s') \in A\text{-rel } S)$
using A
proof (induction arbitrary: s)
case 1: $(A\text{-rel-base } r\ r' s)$
assume $C: (r, r') \in R$ $r' \notin \text{Domain } R$ $(s, r) \in \alpha$
with R **obtain** s' **where** $(s, s') \in S \wedge (s', r') \in \alpha \wedge s' \notin \text{Domain } S$ **by** (blast dest: refinesE)
hence $(s', r') \in \alpha \wedge (s, s') \in A\text{-rel } S$ **by** (blast intro: A-rel-base)
thus $\exists s'. (s', r') \in \alpha \wedge (s, s') \in A\text{-rel } S$ **by** (blast)
next
case $C: (A\text{-rel-step } r\ rh\ r')$
assume $A: (r, rh) \in R$ $(rh, r') \in A\text{-rel } R$ $(s, r) \in \alpha$
with R **obtain** sh **where** STEP: $(sh, rh) \in \alpha \wedge (s, sh) \in S$ **by** (blast dest: refinesE)
with $C.IH$ **obtain** s' **where** $(s', r') \in \alpha \wedge (sh, s') \in A\text{-rel } S$ **by** blast
with STEP **have** $(s', r') \in \alpha \wedge (s, s') \in A\text{-rel } S$ **by** (blast intro: A-rel-step)
thus $\exists s'. (s', r') \in \alpha \wedge (s, s') \in A\text{-rel } S$ **by** (blast)
qed

Property transfer lemma for single-valued abstractions (i.e. abstraction functions)

lemma refines-A-rel-sv: $\llbracket R \leq_{\alpha} S; (r, r') \in A\text{-rel } R; \text{single-valued } (\alpha^{-1}); (s, r) \in \alpha; (s', r') \in \alpha \rrbracket \implies (s, s') \in A\text{-rel } S$ **by** (blast dest: single-valuedD refines-A-rel)

3.3 Extension to reflexive states

Up to now we only defined how to relate initial states to terminating states if the algorithm makes at least one step. In this section, we also add the reflexive part: Initial states for that no steps can be made are mapped to themselves.

definition

$\text{ndet-algo } R == (A\text{-rel } R) \cup \{(s, s) \mid s. s \notin \text{Domain } R\}$

lemma ndet-algo-A-rel: $\llbracket x \in \text{Domain } R; (x, y) \in \text{ndet-algo } R \rrbracket \implies (x, y) \in A\text{-rel } R$ **by** (unfold ndet-algo-def, auto)

lemma ndet-algoE: $\llbracket (s, s') \in \text{ndet-algo } R; \llbracket (s, s') \in A\text{-rel } R \rrbracket \implies P; \llbracket s = s'; s \notin \text{Domain } R \rrbracket \implies P \rrbracket \implies P$ **by** (unfold ndet-algo-def, auto)

lemma ndet-algoE': $\llbracket (s, s') \in \text{ndet-algo } R; \llbracket (s, s') \in A\text{-rel } R; s \in \text{Domain } R; s' \notin \text{Domain } R \rrbracket \implies P; \llbracket s = s'; s \notin \text{Domain } R \rrbracket \implies P \rrbracket \implies P$
using dom-subset[of R] termstate[of $s\ s' R$]

by (auto elim!: ndet-algoE)

ndet-algo is total (i.e. the algorithm is defined for every initial state), if R^{-1} is well founded

lemma ndet-algo-total: wf (R^{-1}) \implies Domain (ndet-algo R) = UNIV
by (unfold ndet-algo-def) (auto simp add: wf-dom-eq)

The result of the algorithm is always a terminating state

lemma termstate-ndet-algo: $(s, s') \in \text{ndet-algo } R \implies s' \notin \text{Domain } R$ **by** (unfold ndet-algo-def, auto dest: termstate)

Property transfer lemma for *ndet-algo*

lemma refines-ndet-algo[rule-format]:
assumes $R: S \leq_{\alpha} R$ and $A: (c, c') \in \text{ndet-algo } S$
shows $\forall a. (a, c) \in \alpha \longrightarrow (\exists a'. (a', c') \in \alpha \wedge (a, a') \in \text{ndet-algo } R)$
proof (intro allI impI)
fix a assume B: $(a, c) \in \alpha$
{ assume CASE: $c \in \text{Domain } S$
with A have $(c, c') \in A\text{-rel } S$ **by** (blast elim: ndet-algoE)
with R B obtain a' where $(a', c') \in \alpha \wedge (a, a') \in A\text{-rel } R$ **by** (blast dest: refines-A-rel)
moreover hence $(a, a') \in \text{ndet-algo } R$ **by** (unfold ndet-algo-def, simp)
ultimately have $\exists a'. (a', c') \in \alpha \wedge (a, a') \in \text{ndet-algo } R$ **by** blast
} moreover {
assume CASE: $c \notin \text{Domain } S$
with A have $c = c'$ **by** (blast elim: ndet-algoE')
moreover have $a \notin \text{Domain } R$ **proof**
assume $a \in \text{Domain } R$
with B R have $c \in \text{Domain } S$ **by** (auto elim: refinesE2)
with CASE show False ..
qed
ultimately have $\exists a'. (a', c') \in \alpha \wedge (a, a') \in \text{ndet-algo } R$ using B **by** (unfold ndet-algo-def, blast)
} ultimately show $\exists a'. (a', c') \in \alpha \wedge (a, a') \in \text{ndet-algo } R$ **by** blast
qed

Property transfer lemma for single-valued abstractions (i.e. Abstraction functions)

lemma refines-ndet-algo-sv: $\llbracket S \leq_{\alpha} R; (c, c') \in \text{ndet-algo } S; \text{single-valued } (\alpha^{-1}); (a, c) \in \alpha; (a', c') \in \alpha \rrbracket \implies (a, a') \in \text{ndet-algo } R$ **by** (blast dest: single-valuedD refines-ndet-algo)

3.4 Well-foundedness

lemma wf-imp-minimal: $\llbracket \text{wf } S; x \in Q \rrbracket \implies \exists z \in Q. (\forall x. (x, z) \in S \longrightarrow x \notin Q)$ **by** (auto iff add: wf-eq-minimal)

This lemma allows to show well-foundedness of a refining relation by providing a well-founded refined relation for each element in the domain of the refining relation.

```

lemma refines-wf:
  assumes A: !!r.  $\llbracket r \in \text{Domain } R \rrbracket \implies (s \ r, r) \in \alpha \ r \wedge R \leq_{\alpha} r \ S \ r \wedge \text{wf } ((S \ r)^{-1})$ 
  shows wf  $(R^{-1})$ 
proof (rule wfI-min)
  fix Q and e :: 'a
  assume NOTEMPTY:  $e \in Q$ 
  moreover {
    assume  $e \notin \text{Domain } R$ 
    hence  $\forall y. (e, y) \in R \implies y \notin Q$  by blast
  } moreover {
    assume C:  $e \in \text{Domain } R$ 
    with A have MAP:  $(s \ e, e) \in \alpha \ e$  and REF:  $R \leq_{\alpha} e \ S \ e$  and WF:  $\text{wf } ((S \ e)^{-1})$ 
    by (auto)
    let ?aQ =  $((\alpha \ e)^{-1})$  “ Q
    from MAP NOTEMPTY have s e ∈ ?aQ by auto
    with WF wf-imp-minimal[of  $(S \ e)^{-1}$ , simplified] have  $\exists z \in ?aQ. (\forall x. (z, x) \in S \ e \implies x \notin ?aQ)$  by auto
    then obtain z where ZMIN:  $z \in ?aQ \wedge (\forall x. (z, x) \in S \ e \implies x \notin ?aQ)$  by blast
    then obtain q where QP:  $(z, q) \in \alpha \ e \wedge q \in Q$  by blast
    have  $\forall x. (q, x) \in R \implies x \notin Q$  proof (intro allI impI)
      fix x
      assume  $(q, x) \in R$ 
      with REF QP obtain xt where ZREF:  $(z, xt) \in S \ e \wedge (xt, x) \in \alpha \ e$  by (blast dest: refinesE)
      with ZMIN have  $xt \notin ?aQ$  by simp
      moreover from ZREF have  $x \in Q \implies xt \in ?aQ$  by blast
      ultimately show  $x \notin Q$  by blast
    qed
    with QP have  $\exists q \in Q. \forall y. (q, y) \in R \implies y \notin Q$  by blast
  } ultimately show  $\exists z \in Q. \forall y. (y, z) \in R^{-1} \implies y \notin Q$  by blast
qed

```

3.4.1 The relations $>$ and \supset on finite domains

```

definition greaterN N ==  $\{(i, j) . j < i \wedge i \leq (N :: \text{nat})\}$ 
definition greaterS S ==  $\{(a, b) . b \subset a \wedge a \subseteq (S :: 'a \text{ set})\}$ 

```

$>$ on initial segment of nat is well founded

```

lemma wf-greaterN: wf (greaterN N)
  apply (unfold greaterN-def)
  apply (rule wf-subset[of measure  $(\lambda k. (N - k))$ ], blast)
  apply (clarify, simp add: measure-def inv-image-def)
done

```

Strict version of card-mono

```

lemma card-mono-strict:  $\llbracket \text{finite } B; A \subset B \rrbracket \implies \text{card } A < \text{card } B$  proof -
  assume F: finite B and S:  $A \subset B$ 
  hence FA: finite A by (auto intro: finite-subset)
  from S obtain x where P:  $x \in B \wedge x \notin A \wedge A - \{x\} = A \wedge \text{insert } x \ A \subseteq B$  by auto

```

```

with FA have card (insert x A) = Suc (card A) by (simp)
moreover from F P have card (insert x A) ≤ card B by (fast intro: card-mono)
ultimately show ?thesis by simp
qed

```

\supset on finite sets is well founded

This is shown here by embedding the \supset relation into the $>$ relation, using cardinality

```

lemma wf-greaterS: finite S ==> wf (greaterS S) proof -
  assume FS: finite S — For this purpose, we show that we can embed greaterS
  into the greaterN by the inverse image of cardinality
  have {(a,b) . b ⊂ a ∧ a ⊆ S} ⊆ inv-image (greaterN (card S)) card proof -
  {
    fix a b
    assume A: b ⊂ a a ⊆ S
    with FS have Fab: finite a finite b by (auto simp add: finite-subset)
    with A FS have card b < card a & card a ≤ card S by (fast intro: card-mono
    card-mono-strict)
    } note R=this
    thus ?thesis by (auto simp add: inv-image-def greaterN-def)
  qed
  thus ?thesis by (unfold greaterS-def, blast intro: wf-greaterN wf-subset)
qed

```

This lemma shows well-foundedness of saturation algorithms, where in each step some set is increased, and this set remains below some finite upper bound

```

lemma sat-wf:
  assumes subset: !!r r'. (r,r') ∈ R ==> α r ⊂ α r' ∧ α r' ⊆ U
  assumes finite: finite U
  shows wf (R⁻¹)
proof -
  have R⁻¹ ⊆ inv-image (greaterS U) α by (auto simp add: inv-image-def greaterS-def
  dest: subset)
  moreover have wf (inv-image (greaterS U) α) using finite by (blast intro:
  wf-greaterS)
  ultimately show ?thesis by (blast intro: wf-subset)
qed

```

3.5 Implementation

The first step to implement a nondeterministic algorithm specified by a relation R is to provide a deterministic refinement w.r.t. the identity abstraction Id . We can describe such a deterministic refinement as the graph of a partial function sel . We call this function a selector function, because it selects the next state from the possible states specified by R .

In order to get a working implementation, we must prove termination. That is, we have to show that $(\text{graph sel})^{-1}$ is well-founded. If we already know that R^{-1} is well-founded, this property transfers to $(\text{graph sel})^{-1}$.

Once obtained well-foundedness, we can use the selector function to implement the following recursive function:

algo $s = \text{case sel } s \text{ of } \text{None} \Rightarrow s \mid \text{Some } s' \Rightarrow \text{algo } s'$

And we can show, that *algo* is consistent with *ndet-algo* R , that is $(s, \text{algo } s) \in \text{ndet-algo } R$.

3.5.1 Graphs of functions

The graph of a (partial) function is the relation of arguments and function values

definition $\text{graph } f == \{(x, x') . f x = \text{Some } x'\}$

```

lemma  $\text{graphI[intro]}: f x = \text{Some } x' \implies (x, x') \in \text{graph } f$  by (unfold graph-def, auto)
lemma  $\text{graphD[dest]}: (x, x') \in \text{graph } f \implies f x = \text{Some } x'$  by (unfold graph-def, auto)
lemma  $\text{graph-dom-iff1}: (x \notin \text{Domain } (\text{graph } f)) = (f x = \text{None})$  by (cases f x) auto
lemma  $\text{graph-dom-iff2}: (x \in \text{Domain } (\text{graph } f)) = (f x \neq \text{None})$  by (cases f x) auto

```

3.5.2 Deterministic refinement w.r.t. the identity abstraction

```

lemma  $\text{detRef-eq}: (\text{graph sel} \leq_{Id} R) = ((\forall s s'. \text{sel } s = \text{Some } s' \longrightarrow (s, s') \in R) \wedge
(\forall s. \text{sel } s = \text{None} \longrightarrow s \notin \text{Domain } R))$ 
by (unfold refines-def) (auto iff add: graph-dom-iff2)

```

```

lemma  $\text{detRef-wf-transfer}: [\text{wf } (R^{-1}); \text{graph sel} \leq_{Id} R] \implies \text{wf } ((\text{graph sel})^{-1})$ 
by (rule refines-wf[where s=id and  $\alpha=\lambda x. \text{Id}$  and  $S=\lambda x. R$ ]) simp

```

3.5.3 Recursive characterization

```

locale  $\text{detRef-impl} =$ 
  fixes algo and sel and R
  assumes  $\text{detRef}: \text{graph sel} \leq_{Id} R$ 
  assumes  $\text{algo-rec[simp]}: !! s s'. \text{sel } s = \text{Some } s' \implies \text{algo } s = \text{algo } s' \text{ and}$ 
   $\text{algo-term[simp]}: !! s. \text{sel } s = \text{None} \implies \text{algo } s = s$ 
  assumes  $\text{wf}: \text{wf } ((\text{graph sel})^{-1})$ 

lemma (in detRef-impl) sel-cons:
   $\text{sel } s = \text{Some } s' \implies (s, s') \in R$ 
   $\text{sel } s = \text{None} \implies s \notin \text{Domain } R$ 
   $s \in \text{Domain } R \implies \exists s'. \text{sel } s = \text{Some } s'$ 
   $s \notin \text{Domain } R \implies \text{sel } s = \text{None}$ 
  using detRef
  by (simp-all only: detRef-eq) (cases sel s, blast, blast) +

```

```

lemma (in detRef-impl) algo-correct: (s,algo s) ∈ ndet-algo R proof -
{
  assume C: s ∈ Domain R
  have !!s. s ∈ Domain R → (s,algo s) ∈ A-rel R
  proof (rule wf-induct[OF wf, of λs. s ∈ Domain R → (s,algo s) ∈ A-rel R]; intro
  impI)
    fix s
    assume A: s ∈ Domain R and IH: ∀ y. (y, s) ∈ (graph sel)-1 → y ∈ Domain
    R → (y, algo y) ∈ A-rel R
    then obtain sh where SH: sel s = Some sh ∧ (s,sh) ∈ R using sel-cons by
    blast
    hence AS: algo s = algo sh by auto
    {
      assume C: sh ∉ Domain R
      hence sel sh = None by (auto dest: sel-cons)
      hence algo sh = sh by (auto)
      moreover from SH C have (s,sh) ∈ A-rel R by (blast intro: A-rel-base)
      ultimately have (s,algo s) ∈ A-rel R using AS by simp
    } moreover {
      assume C: sh ∈ Domain R
      with SH IH AS A have (sh,algo s) ∈ A-rel R by auto
      with SH have (s,algo s) ∈ A-rel R by (blast intro: A-rel-step)
    } ultimately show (s,algo s) ∈ A-rel R by blast
  qed
  with C have (s,algo s) ∈ A-rel R by simp
  hence ?thesis by (unfold ndet-algo-def, auto)
} moreover {
  assume C: s ∉ Domain R
  hence s = algo s by (auto dest: sel-cons)
  with C have ?thesis by (unfold ndet-algo-def, auto)
} ultimately show ?thesis by blast
qed
end

```

4 Dynamic pushdown networks

```

theory DPN
imports DPN-Setup SRS FSM NDET
begin

```

Dynamic pushdown networks (DPNs) are a model for parallel, context free processes where processes can create new processes.

They have been introduced in [1]. In this theory we formalize DPNs and the automata based algorithm for calculating a representation of the (regular) set of backward reachable configurations, starting at a regular set of configurations.

We describe the algorithm nondeterministically, and prove its termination and correctness.

4.1 Dynamic pushdown networks

4.1.1 Definition

```
record ('c,'l) DPN-rec =
  csyms :: 'c set
  ssyms :: 'c set
  sep :: 'c
  labels :: 'l set
  rules :: ('c,'l) SRS
```

A dynamic pushdown network consists of a finite set of control symbols, a finite set of stack symbols, a separator symbol¹, a finite set of labels and a finite set of labelled string rewrite rules.

The set of control and stack symbols are disjoint, and both do not contain the separator. A string rewrite rule is either of the form $[p,\gamma] \rightarrow_a p1\#w1$ or $[p,\gamma] \rightarrow_a p1\#w1@\#\#p2\#w2$ where $p,p1,p2$ are control symbols, $w1,w2$ are sequences of stack symbols, a is a label and $\#$ is the separator.

```
locale DPN =
  fixes M
  fixes separator (#)
  defines sep-def: # == sep M
  assumes sym-finite: finite (csyms M) finite (ssyms M)
  assumes sym-disjoint: csyms M ∩ ssyms M = {} # ∉ csyms M ∪ ssyms M
  assumes lab-finite: finite (labels M)
  assumes rules-finite: finite (rules M)
  assumes rule-fmt: r ∈ rules M ==>
    (exists p γ a p' w. p ∈ csyms M ∧ γ ∈ ssyms M ∧ p' ∈ csyms M ∧ w ∈ lists (ssyms M)
    ∧ a ∈ labels M ∧ r = p#[γ] →_a p'#w)
    ∨ (exists p γ a p1 w1 p2 w2. p ∈ csyms M ∧ γ ∈ ssyms M ∧ p1 ∈ csyms M ∧ w1 ∈ lists (ssyms M)
    ∧ p2 ∈ csyms M ∧ w2 ∈ lists (ssyms M) ∧ a ∈ labels M ∧ r = p#[γ] →_a
    p1#w1@\#\#p2#w2)

  lemma (in DPN) sep-fold: sep M == # by (simp add: sep-def)

  lemma (in DPN) sym-disjoint': sep M ∉ csyms M ∪ ssyms M using sym-disjoint
  by (simp add: sep-def)
```

4.1.2 Basic properties

```
lemma (in DPN) syms-part: x ∈ csyms M ==> x ∉ ssyms M x ∈ ssyms M ==> x ∉ csyms M using sym-disjoint by auto
lemma (in DPN) syms-sep: # ∉ csyms M # ∉ ssyms M using sym-disjoint by auto
```

¹In the final version of [1], no separator symbols are used. We use them here because we think it simplifies formalization of the proofs.

```

lemma (in DPN) syms-sep': sep M∉csyms M sep M∉ssyms M using syms-sep
by (auto simp add: sep-def)

lemma (in DPN) rule-cases[consumes 1, case-names no-spawn spawn]:
assumes A: r∈rules M
assumes NOSPAWN: !! p γ a p' w. [p∈csyms M; γ∈ssyms M; p'∈csyms M;
w∈lists (ssyms M); a∈labels M; r=p#[γ] ↪a p'#w]  $\implies$  P
assumes SPAWN: !! p γ a p1 w1 p2 w2. [p∈csyms M; γ∈ssyms M; p1∈csyms
M; w1∈lists (ssyms M); p2∈csyms M; w2∈lists (ssyms M); a∈labels M; r=p#[γ]
↪a p1#w1@#p2#w2]  $\implies$  P
shows P
using A NOSPAWN SPAWN
by (blast dest!: rule-fmt)

lemma (in DPN) rule-cases':
[ $\exists$  r∈rules M;
!! p γ a p' w. [p∈csyms M; γ∈ssyms M; p'∈csyms M; w∈lists (ssyms M);
a∈labels M; r=p#[γ] ↪a p'#w]  $\implies$  P;
!! p γ a p1 w1 p2 w2. [p∈csyms M; γ∈ssyms M; p1∈csyms M; w1∈lists (ssyms
M); p2∈csyms M; w2∈lists (ssyms M); a∈labels M; r=p#[γ] ↪a p1#w1@(sep
M)#p2#w2]  $\implies$  P]
 $\implies$  P by (unfold sep-fold) (blast elim!: rule-cases)

lemma (in DPN) rule-prem-fmt: r∈rules M  $\implies$   $\exists$  p γ a c'. p∈csyms M  $\wedge$  γ∈ssyms
M  $\wedge$  a∈labels M  $\wedge$  set c' ⊆ csyms M  $\cup$  ssyms M  $\cup$  {#}  $\wedge$  r=(p#[γ] ↪a c')
apply (erule rule-cases)
by (auto)

lemma (in DPN) rule-prem-fmt': r∈rules M  $\implies$   $\exists$  p γ a c'. p∈csyms M  $\wedge$ 
γ∈ssyms M  $\wedge$  a∈labels M  $\wedge$  set c' ⊆ csyms M  $\cup$  ssyms M  $\cup$  {sep M}  $\wedge$  r=(p#[γ]
↪a c') by (unfold sep-fold, rule rule-prem-fmt)

lemma (in DPN) rule-prem-fmt2: [p,γ]↪a c' ∈ rules M  $\implies$  p∈csyms M  $\wedge$ 
γ∈ssyms M  $\wedge$  a∈labels M  $\wedge$  set c' ⊆ csyms M  $\cup$  ssyms M  $\cup$  {#} by (fast dest:
rule-prem-fmt)
lemma (in DPN) rule-prem-fmt2': [p,γ]↪a c' ∈ rules M  $\implies$  p∈csyms M  $\wedge$ 
γ∈ssyms M  $\wedge$  a∈labels M  $\wedge$  set c' ⊆ csyms M  $\cup$  ssyms M  $\cup$  {sep M} by (unfold
sep-fold, rule rule-prem-fmt2)

lemma (in DPN) rule-fmt-fs: [p,γ]↪a p'#c' ∈ rules M  $\implies$  p∈csyms M  $\wedge$  γ∈ssyms
M  $\wedge$  a∈labels M  $\wedge$  p'∈csyms M  $\wedge$  set c' ⊆ csyms M  $\cup$  ssyms M  $\cup$  {#}
apply (erule rule-cases)
by (auto)

```

4.1.3 Building DPNs

Sanity check: we can create valid DPNs by adding rules to an empty DPN

definition dpn-empty C S s ≡ ()
csyms = C,

```

ssyms = S,
sep = s,
labels = {},
rules = {}
)

definition dpn-add-local-rule p γ a p1 w1 D ≡ D ( labels := insert a (labels D),
rules := insert ([p,γ],a,p1#w1) (rules D) )

definition dpn-add-spawn-rule p γ a p1 w1 p2 w2 D ≡ D ( labels := insert a (labels D),
rules := insert ([p,γ],a,p1#w1@sep D#p2#w2) (rules D) )

lemma dpn-empty-invar[simp]: [|finite C; finite S; C ∩ S = {}; s ∉ C ∪ S|] ==> DPN
(dpn-empty C S s)
  apply unfold-locales unfolding dpn-empty-def by auto

lemma dpn-add-local-rule-invar[simp]:
  assumes A: {p,p1} ⊆ csyms D insert γ (set w1) ⊆ ssyms D and DPN D
  shows DPN (dpn-add-local-rule p γ a p1 w1 D)
proof -
  interpret DPN D sep D by fact
  show ?thesis
    unfolding dpn-add-local-rule-def
    apply unfold-locales
    using sym-finite sym-disjoint lab-finite rules-finite
    apply simp-all
    apply (erule disjE)
    subgoal for r using A by auto
    subgoal for r using rule-fmt[of r] by metis
    done
qed

lemma dpn-add-spawn-rule-invar[simp]:
  assumes A: {p,p1,p2} ⊆ csyms D insert γ (set w1 ∪ set w2) ⊆ ssyms D and
DPN D
  shows DPN (dpn-add-spawn-rule p γ a p1 w1 p2 w2 D)
proof -
  interpret DPN D sep D by fact
  show ?thesis
    unfolding dpn-add-spawn-rule-def
    apply unfold-locales
    using sym-finite sym-disjoint lab-finite rules-finite
    apply (simp-all)
    apply (erule disjE)
    subgoal for r apply (rule disjI2) using A apply clarsimp by (metis in-listsI
subset-eq)
    subgoal for r using rule-fmt[of r] by metis
    done
qed

```

4.2 M-automata

We are interested in calculating the predecessor sets of regular sets of configurations. For this purpose, the regular sets of configurations are represented as finite state machines, that conform to certain constraints, depending on the underlying DPN. These FSMs are called M-automata.

4.2.1 Definition

```
record ('s,'c) MFSM-rec = ('s,'c) FSM-rec +
  sstates :: 's set
  cstates :: 's set
  sp :: 's ⇒ 'c ⇒ 's
```

M-automata are FSMs whose states are partitioned into control and stack states. For each control state s and control symbol p , there is a unique and distinguished stack state $sp A s p$, and a transition $(s,p,sp A s p) \in \delta$. The initial state is a control state, and the final states are all stack states. Moreover, the transitions are restricted: The only incoming transitions of control states are separator transitions from stack states. The only outgoing transitions are the $(s,p,sp A s p) \in \delta$ transitions mentioned above. The $sp A s p$ -states have no other incoming transitions.

```
locale MFSM = DPN M + FSM A
  for M A +
    assumes alpha-cons:  $\Sigma A = csyms M \cup ssyms M \cup \{\#\}$ 
    assumes states-part:  $sstates A \cap cstates A = \{\}$ 
    assumes uniqueSp:  $\llbracket s \in cstates A; p \in csyms M \rrbracket \implies sp A s p \in sstates A \llbracket p \in csyms M; p' \in csyms M; s \in cstates A; s' \in cstates A; sp A s p = sp A s' p' \rrbracket \implies s = s' \wedge p = p'$ 
    assumes delta-fmt:  $\delta A \subseteq (sstates A \times ssyms M \times (sstates A - \{sp A s p \mid s p . s \in cstates A \wedge p \in csyms M\})) \cup (sstates A \times \{\#\} \times cstates A) \cup \{(s,p,sp A s p) \mid s p . s \in cstates A \wedge p \in csyms M\}$ 
    assumes s0-fmt:  $s0 A \in cstates A$ 
    assumes F-fmt:  $F A \subseteq sstates A$  — This deviates slightly from [1], as we cannot represent the empty configuration here. However, this restriction is harmless, since the only predecessor of the empty configuration is the empty configuration itself.
    constrains M::('c,'l,'e1) DPN-rec-scheme
    constrains A::('s,'c,'e2) MFSM-rec-scheme
```

```
lemma (in MFSM) alpha-cons':  $\Sigma A = csyms M \cup ssyms M \cup \{sep M\}$  by
  (unfold sep-fold, rule alpha-cons)
lemma (in MFSM) delta-fmt':  $\delta A \subseteq (sstates A \times ssyms M \times (sstates A - \{sp A s p \mid s p . s \in cstates A \wedge p \in csyms M\})) \cup (sstates A \times \{sep M\} \times cstates A)$ 
```

$\cup \{(s, p, sp A s p) \mid s p . s \in cstates A \wedge p \in csyms M\}$
 $\delta A \supseteq \{(s, p, sp A s p) \mid s p . s \in cstates A \wedge p \in csyms M\}$ by
 $(unfold sep-fold, (rule delta-fmt)+)$

4.2.2 Basic properties

lemma (in MFSM) finite-cs-states: $finite(sstates A) finite(cstates A)$

proof –

have $sstates A \subseteq Q A \wedge cstates A \subseteq Q A$ by (auto simp add: states-part)

then show $finite(sstates A) finite(cstates A)$ by (auto dest: finite-subset intro: finite-states)

qed

lemma (in MFSM) sep-out-syms: $x \in csyms M \Rightarrow x \neq \sharp x \in ssyms M \Rightarrow x \neq \sharp$
by (auto iff add: syms-sep)

lemma (in MFSM) sepI: $\llbracket x \in \Sigma A; x \notin csyms M; x \notin ssyms M \rrbracket \Rightarrow x = \sharp$ using alpha-cons by auto

lemma (in MFSM) sep-out-syms': $x \in csyms M \Rightarrow x \neq sep M x \in ssyms M \Rightarrow x \neq sep M$ by (unfold sep-fold, (fast dest: sep-out-syms) +)

lemma (in MFSM) sepI': $\llbracket x \in \Sigma A; x \notin csyms M; x \notin ssyms M \rrbracket \Rightarrow x = sep M$ using alpha-cons' by auto

lemma (in MFSM) states-partI1: $x \in sstates A \Rightarrow \neg x \in cstates A$ using states-part by (auto)

lemma (in MFSM) states-partI2: $x \in cstates A \Rightarrow \neg x \in sstates A$ using states-part by (auto)

lemma (in MFSM) states-part-elim[elim]: $\llbracket q \in Q A; q \in sstates A \Rightarrow P; q \in cstates A \Rightarrow P \rrbracket \Rightarrow P$ using states-part by (auto)

lemmas (in MFSM) mfsm-cons = sep-out-syms sepI sep-out-syms' sepI' states-partI1 states-partI2 syms-part syms-sep uniqueSp

lemmas (in MFSM) mfsm-cons' = sep-out-syms sepI sep-out-syms' sepI' states-partI1 states-partI2 syms-part uniqueSp

lemma (in MFSM) delta-cases: $\llbracket (q, p, q') \in \delta A; q \in sstates A \wedge p \in ssyms M \wedge q' \in sstates A \wedge q' \notin \{sp A s p \mid s p . s \in cstates A \wedge p \in csyms M\} \rrbracket \Rightarrow P;$

$q \in sstates A \wedge p = \sharp \wedge q' \in cstates A \Rightarrow P;$

$q \in cstates A \wedge p \in csyms M \wedge q' = sp A q p \Rightarrow P$

$\rrbracket \Rightarrow P$

using delta-fmt by auto

lemma (in MFSM) delta-elems: $(q, p, q') \in \delta A \Rightarrow q \in sstates A \wedge ((p \in ssyms M \wedge q' \in sstates A \wedge (q' \notin \{sp A s p \mid s p . s \in cstates A \wedge p \in csyms M\})) \vee (p = \sharp \wedge q' \in cstates A)) \vee (q \in cstates A \wedge p \in csyms M \wedge q' = sp A q p)$

using delta-fmt by auto

lemma (in MFSM) delta-cases': $\llbracket (q, p, q') \in \delta A; q \in sstates A \wedge p \in ssyms M \wedge q' \in sstates A \wedge q' \notin \{sp A s p \mid s p . s \in cstates A \wedge p \in csyms M\} \rrbracket \Rightarrow P;$

$q \in sstates A \wedge p = sep M \wedge q' \in cstates A \Rightarrow P;$

$q \in cstates A \wedge p = sep M \wedge q' = sp A q p \Rightarrow P$

$q \in cstates A \wedge p \in csyms M \wedge q' = sp A q p \implies P \implies P$
using delta-fmt' by auto

lemma (in MFSM) delta-elems': $(q,p,q') \in \delta A \implies q \in sstates A \wedge ((p \in ssyms M \wedge q' \in sstates A \wedge (q' \notin \{sp A s p \mid s p . s \in cstates A \wedge p \in csyms M\})) \vee (p = sep M \wedge q' \in cstates A)) \vee (q \in cstates A \wedge p \in csyms M \wedge q' = sp A q p)$
using delta-fmt' by auto

4.2.3 Some implications of the M-automata conditions

This list of properties is taken almost literally from [1].

Each control state s has $sp A s p$ as its unique p -successor

lemma (in MFSM) cstate-succ-ex: $\llbracket p \in csyms M; s \in cstates A \rrbracket \implies (s, p, sp A s p) \in \delta A$
using delta-fmt by (auto)

lemma (in MFSM) cstate-succ-ex': $\llbracket p \in csyms M; s \in cstates A; \delta A \subseteq D \rrbracket \implies (s, p, sp A s p) \in D$ **using cstate-succ-ex by auto**

lemma (in MFSM) cstate-succ-unique: $\llbracket s \in cstates A; (s, p, x) \in \delta A \rrbracket \implies p \in csyms M \wedge x = sp A s p$ **by (auto elim: delta-cases dest: mfsm-cons')**

Transitions labeled with control symbols only leave from control states

lemma (in MFSM) csym-from-cstate: $\llbracket (s, p, s') \in \delta A; p \in csyms M \rrbracket \implies s \in cstates A$ **by (auto elim: delta-cases dest: mfsm-cons')**

s is the only predecessor of $sp A s p$

lemma (in MFSM) sp-pred-ex: $\llbracket s \in cstates A; p \in csyms M \rrbracket \implies (s, p, sp A s p) \in \delta A$ **using delta-fmt by auto**

lemma (in MFSM) sp-pred-unique: $\llbracket s \in cstates A; p \in csyms M; (s', p', sp A s p) \in \delta A \rrbracket \implies s' = s \wedge p' = p \wedge s' \in cstates A \wedge p' \in csyms M$ **by (erule delta-cases) (auto dest: mfsm-cons')**

Only separators lead from stack states to control states

lemma (in MFSM) sep-in-between: $\llbracket s \in sstates A; s' \in cstates A; (s, p, s') \in \delta A \rrbracket \implies p = \#$ **by (auto elim: delta-cases dest: mfsm-cons')**

lemma (in MFSM) sep-to-cstate: $\llbracket (s, \#, s') \in \delta A \rrbracket \implies s \in sstates A \wedge s' \in cstates A$ **by (auto elim: delta-cases dest: mfsm-cons')**

Stack states do not have successors labelled with control symbols

lemma (in MFSM) sstate-succ: $\llbracket s \in sstates A; (s, \gamma, s') \in \delta A \rrbracket \implies \gamma \notin csyms M$ **by (auto elim: delta-cases dest: mfsm-cons')**

lemma (in MFSM) sstate-succ2: $\llbracket s \in sstates A; (s, \gamma, s') \in \delta A; \gamma \neq \# \rrbracket \implies \gamma \in ssyms M \wedge s' \in sstates A$ **by (auto elim: delta-cases dest: mfsm-cons')**

M-automata do not accept the empty word

```

lemma (in MFSM) not-empty[iff]: []notin lang A
  apply (unfold lang-def langs-def)
  apply (clar simp)
  apply (insert s0-fmt F-fmt)
  apply (subgoal-tac s0 A = f)
  apply (auto dest: fsm-cons')
done

```

The paths through an M-automata have a very special form: Paths starting at a stack state are either labelled entirely with stack symbols, or have a prefix labelled with stack symbols followed by a separator

```

lemma (in MFSM) path-from-sstate: !!s . [s in sstates A; (s,w,f) in trclA A] ==>
  
$$(f \in sstates A \wedge w \in lists (ssyms M)) \vee (\exists w1 w2 t. w = w1 @ \# w2 \wedge w1 \in lists (ssyms M) \wedge t \in sstates A \wedge (s, w1, t) \in trclA A \wedge (t, \# w2, f) \in trclA A)$$

  proof (induct w)
    case Nil thus ?case by (subgoal-tac s=f) auto
  next
    case (Cons e w)
    note IHP[rule-format]=this
    then obtain s' where STEP: (s,e,s') in (delta A) \wedge s in Q A \wedge e in Sigma A \wedge (s',w,f) in trclA A by (fast dest: trclAD-uncons)
    show ?case proof (cases e=?)
      assume e=?
      with IHP have e#w=[]@#w \wedge [] in lists (ssyms M) \wedge s in sstates A \wedge (s,[],s) in trclA A \wedge (s,e#w,f) in trclA A using states-part by (auto)
      thus ?case by force
    next
      assume e=?
      with IHP STEP sstate-succ2 have EC: e in ssyms M \wedge s' in sstates A by blast
      with IHP STEP have (f in sstates A \wedge w in lists (ssyms M)) \vee (\exists w1 w2 t. w = w1 @ \# w2 \wedge w1 in lists (ssyms M) \wedge t in sstates A \wedge (s', w1, t) in trclA A \wedge (t, \# w2, f) in trclA A) (is ?C1 \vee ?C2) by auto
      moreover {
        assume ?C1
        with EC have f in sstates A \wedge e#w in lists (ssyms M) by auto
      } moreover {
        assume ?C2
        then obtain w1 w2 t where CASE: w = w1 @ \# w2 \wedge w1 in lists (ssyms M) \wedge t in sstates A \wedge (s', w1, t) in trclA A \wedge (t, \# w2, f) in trclA A by (fast)
        with EC have e#w = (e#w1) @ \# w2 \wedge e#w1 in lists (ssyms M) by auto
        moreover from CASE STEP IHP have (s, e#w1, t) in trclA A using states-part by auto
        moreover note CASE
        ultimately have \exists w1 w2 t. e#w = w1 @ \# w2 \wedge w1 in lists (ssyms M) \wedge t in sstates A \wedge (s, w1, t) in trclA A \wedge (t, \# w2, f) in trclA A by fast
        } ultimately show ?case by blast
      qed
    qed

```

Using *MFSM.path-from-sstate*, we can describe the format of paths from control states, too. A path from a control state s to some final state starts with a transition $(s, p, sp A s p)$ for some control symbol p . It then continues with a sequence of transitions labelled by stack symbols. It then either ends or continues with a separator transition, bringing it to a control state again, and some further transitions from there on.

```

lemma (in MFSM) path-from-cstate:
  assumes  $A: s \in cstates A \ (s, c, f) \in trclA \ A \ f \in sstates A$ 
  assumes SINGLE:  $!! p \ w . \ [c=p\#w; \ p \in csyms M; \ w \in lists (ssyms M); \ (s, p, sp A s p) \in \delta A; \ (sp A s p, w, f) \in trclA A] \implies P$ 
  assumes CONC:  $!! p \ w \ cr \ t \ s' . \ [c=p\#w@#\#cr; \ p \in csyms M; \ w \in lists (ssyms M); \ t \in sstates A; \ s' \in cstates A; \ (s, p, sp A s p) \in \delta A; \ (sp A s p, w, t) \in trclA A; \ (t, \#, s') \in \delta A; \ (s', cr, f) \in trclA A] \implies P$ 
  shows  $P$ 
  proof (cases c)
    case Nil thus P using A by (subgoal-tac s=f, auto dest: fsm-cons')
  next
    case (Cons p w) note CFMT=this
    with cstate-succ-unique A have SPLIT:  $p \in csyms M \wedge (s, p, sp A s p) \in \delta A \wedge (sp A s p, w, f) \in trclA A$  by (blast dest: trclAD-uncons)
    with path-from-sstate A CFMT uniqueSp have CASES:  $(f \in sstates A \wedge w \in lists (ssyms M)) \vee (\exists w1 w2 t. \ w=w1@#\#w2 \wedge w1 \in lists (ssyms M) \wedge t \in sstates A \wedge (sp A s p, w1, t) \in trclA A \wedge (t, \#, w2, f) \in trclA A)$  (is ?C1  $\vee$  ?C2) by blast
    moreover {
      assume CASE: ?C1
      with SPLIT SINGLE A CFMT have P by fast
    } moreover {
      assume CASE: ?C2
      then obtain w1 w2 t where WFMT:  $w=w1@#\#w2 \wedge w1 \in lists (ssyms M) \wedge t \in sstates A \wedge (sp A s p, w1, t) \in trclA A \wedge (t, \#, w2, f) \in trclA A$  by fast
      with sep-to-cstate obtain s' where  $s' \in cstates A \wedge (t, \#, s') \in \delta A \wedge (s', w2, f) \in trclA A$  by (fast dest: trclAD-uncons)
      with SPLIT CASE WFMT have  $p\#w=p\#w1@#\#w2 \wedge p \in csyms M \wedge w1 \in lists (ssyms M) \wedge t \in sstates A \wedge s' \in cstates A \wedge (s, p, sp A s p) \in \delta A \wedge (sp A s p, w1, t) \in trclA A \wedge (t, \#, s') \in \delta A \wedge (s', w2, f) \in trclA A$  by auto
      with CFMT CONC have P by (fast)
    } ultimately show P by blast
  qed

```

4.3 pre^* -sets of regular sets of configurations

Given a regular set L of configurations and a set Δ of string rewrite rules, $pre^* \Delta L$ is the set of configurations that can be rewritten to some configuration in L , using rules from Δ arbitrarily often.

We first define this set inductively based on rewrite steps, and then provide the characterization described above as a lemma.

```

inductive-set  $pre\text{-star} :: ('c, 'l) \ SRS \Rightarrow ('s, 'c, 'e) \ FSM\text{-rec-scheme} \Rightarrow 'c \ list \ set$ 

```

```


```


$$(pre^*)$$

$$\text{for } \Delta L$$

$$\text{where}$$

$$\text{pre-refl: } c \in \text{lang } L \implies c \in \text{pre}^* \Delta L \mid$$

$$\text{pre-step: } \llbracket c' \in \text{pre}^* \Delta L; (c, a, c') \in \text{tr } \Delta \rrbracket \implies c \in \text{pre}^* \Delta L$$


```



Alternative characterization of  $\text{pre}^* \Delta L$



lemma  $\text{pre-star-alt}$ :  $\text{pre}^* \Delta L = \{c . \exists c' \in \text{lang } L . \exists as . (c, as, c') \in \text{trcl } (\text{tr } \Delta)\}$



proof –



```


$$\{$$

$$\text{fix } x c' as$$

$$\text{have } \llbracket x \hookrightarrow_{as} c' \in \text{trcl } (\text{tr } \Delta); c' \in \text{lang } L \rrbracket \implies x \in \text{pre}^* \Delta L$$

$$\text{by (induct rule: trcl.induct) (auto intro: pre-step pre-refl)}$$

$$\}$$

$$\text{then show } ?thesis$$

$$\text{by (auto elim!: pre-star.induct intro: trcl.intros)}$$


```



qed


```

lemma pre-star-altI : $\llbracket c' \in \text{lang } L; c \hookrightarrow_{as} c' \in \text{trcl } (\text{tr } \Delta) \rrbracket \implies c \in \text{pre}^* \Delta L$ **by** (unfold pre-star-alt , auto)

lemma pre-star-altE : $\llbracket c \in \text{pre}^* \Delta L; !!c' as. \llbracket c' \in \text{lang } L; c \hookrightarrow_{as} c' \in \text{trcl } (\text{tr } \Delta) \rrbracket \implies P \rrbracket \implies P$ **by** (unfold pre-star-alt , auto)

4.4 Nondeterministic algorithm for pre^*

In this section, we formalize the saturation algorithm for computing $\text{pre}^* \Delta L$ from [1]. Roughly, the algorithm works as follows:

1. Set $D = \delta A$
2. Choose a rule $([p, \gamma], a, c') \in \text{rules } M$ and states $q, q' \in Q A$, such that D can read the configuration c' from state q and end in state q' (i.e. $(q, c', q') \in \text{trclAD } A D$) and such that $(sp A q p, \gamma, q') \notin D$. If this is not possible, terminate.
3. Add the transition $(sp A q p, \gamma, q') \notin D$ to D and continue with step 2

Intuitively, the behaviour of this algorithm can be explained as follows: If there is a configuration $c_1 @ c' @ c_2 \in \text{pre}^* \Delta L$, and a rule $(p \# \gamma, a, c') \in \Delta$, then we also have $c_1 @ p \# \gamma @ c_2 \in \text{pre}^* \Delta L$. The effect of step 3 is exactly adding these configurations $c_1 @ p \# \gamma @ c_2$ to the regular set of configurations.

We describe the algorithm nondeterministically by its step relation $ps\text{-}R$. Each step describes the addition of one transition.

In this approach, we directly restrict the domain of the step-relation to transition relations below some upper bound $ps\text{-}upper$. We will later show,

that the initial transition relation of an M-automata is below this upper bound, and that the step-relation preserves the property of being below this upper bound.

We define $ps\text{-upper } M A$ as a finite set, and show that the initial transition relation δA of an M-automata is below $ps\text{-upper } M A$, and that $ps\text{-R } M A$ preserves the property of being below the finite set $ps\text{-upper } M A$. Note that we use the more fine-grained $ps\text{-upper } M A$ as upper bound for the termination proof rather than $Q A \times \Sigma A \times Q A$, as $sp A q p$ is only specified for control states q and control symbols p . Hence we need the finer structure of $ps\text{-upper } M A$ to guarantee that sp is only applied to arguments it is specified for. Anyway, the fine-grained $ps\text{-upper } M A$ bound is also needed for the correctness proof.

definition $ps\text{-upper} :: ('c,'l,'e1) DPN\text{-rec-scheme} \Rightarrow ('s,'c,'e2) MFSM\text{-rec-scheme} \Rightarrow ('s,'c) LTS$ **where**

$ps\text{-upper } M A == (sstates A \times ssyms M \times sstates A) \cup (sstates A \times \{sep M\} \times cstates A) \cup \{(s,p,sp A s p) \mid s \in cstates A \wedge p \in csyms M\}$

inductive-set $ps\text{-R} :: ('c,'l,'e1) DPN\text{-rec-scheme} \Rightarrow ('s,'c,'e2) MFSM\text{-rec-scheme} \Rightarrow (('s,'c) LTS * ('s,'c) LTS) set$ **for** $M A$

where

$\llbracket [p,\gamma] \xrightarrow{a} c' \in rules M; (q,c',q') \in trclAD A D; (sp A q p, \gamma, q') \notin D; D \subseteq ps\text{-upper } M A \rrbracket \implies (D, insert (sp A q p, \gamma, q') D) \in ps\text{-R } M A$

lemma $ps\text{-R-dom-below}: (D,D') \in ps\text{-R } M A \implies D \subseteq ps\text{-upper } M A$ **by** (auto elim: $ps\text{-R.cases}$)

4.4.1 Termination

Termination of our algorithm is equivalent to well-foundedness of its (converse) step relation, that is, we have to show $wf((ps\text{-R } M A)^{-1})$.

In the following, we also establish some properties of transition relations below $ps\text{-upper } M A$, that will be used later in the correctness proof.

lemma (in MFSM) $ps\text{-upper-cases}: \llbracket (s,e,s') \in ps\text{-upper } M A; [s \in sstates A; e \in ssyms M; s' \in sstates A] \implies P; [s \in sstates A; e = \#; s' \in cstates A] \implies P; [s \in cstates A; e \in csyms M; s' = sp A s e] \implies P \rrbracket \implies P$
by (unfold $ps\text{-upper-def}$ $sep\text{-def}$, auto)

lemma (in MFSM) $ps\text{-upper-cases}': \llbracket (s,e,s') \in ps\text{-upper } M A; [s \in sstates A; e \in ssyms M; s' \in sstates A] \implies P; [s \in sstates A; e = sep M; s' \in cstates A] \implies P; [s \in cstates A; e \in csyms M; s' = sp A s e] \implies P \rrbracket \implies P$
apply (rule $ps\text{-upper-cases}$)
by (unfold $sep\text{-def}$) auto

lemma (in MFSM) ps-upper-below-trivial: $ps\text{-upper } M A \subseteq Q A \times \Sigma A \times Q A$
by (*unfold ps-upper-def*, *auto simp add: states-part alpha-cons uniqueSp sep-def*)

lemma (in MFSM) ps-upper-finite: $finite (ps\text{-upper } M A)$ **using** *ps-upper-below-trivial finite-delta-dom* **by** (*auto simp add: finite-subset*)

The initial transition relation of the M-automaton is below $ps\text{-upper } M A$

lemma (in MFSM) initial-delta-below: $\delta A \subseteq ps\text{-upper } M A$ **using** *delta-fmt* **by** (*unfold ps-upper-def sep-def*) *auto*

Some lemmas about structure of transition relations below $ps\text{-upper } M A$

lemma (in MFSM) cstate-succ-unique': $\llbracket s \in cstates A; (s, p, x) \in D; D \subseteq ps\text{-upper } M A \rrbracket \implies p \in csyms M \wedge x = sp A s p$ **by** (*auto elim: ps-upper-cases dest: mfsm-cons'*)
lemma (in MFSM) csym-from-cstate': $\llbracket (s, p, s') \in D; D \subseteq ps\text{-upper } M A; p \in csyms M \rrbracket \implies s \in cstates A$ **by** (*auto elim: ps-upper-cases dest: mfsm-cons'*)

The only way to end up in a control state is after executing a separator.

lemma (in MFSM) ctrl-after-sep: **assumes** *BELOW*: $D \subseteq ps\text{-upper } M A$
assumes $A: (q, c', q') \in trclAD A D \quad c' \neq \emptyset$
shows $q' \in cstates A = (last c' = \#)$
proof –
from A **have** $(q, butlast c' @ [last c'], q') \in trclAD A D$ **by** *auto*
with A **obtain** qh **where** $(qh, [last c'], q') \in trclAD A D$ **by** (*blast dest: trclAD-unconcat*)
hence $(qh, last c', q') \in D$ **by** (*fast dest: trclAD-single*)
with *BELOW* **have** *IS*: $(qh, last c', q') \in ps\text{-upper } M A$ **by** *fast*
thus *?thesis* **by** (*erule-tac ps-upper-cases*) (*auto dest: mfsm-cons' simp add: sep-out-syms*)
qed

When applying a rules right hand side to a control state, we will get to a stack state

lemma (in MFSM) ctrl-rule: **assumes** *BELOW*: $D \subseteq ps\text{-upper } M A$
assumes $A: ([p, \gamma], a, c') \in rules M$ **and** $B: q \in cstates A \quad (q, c', q') \in trclAD A D$
shows $q' \in sstates A$
proof –
from A **show** *?thesis*
proof (*cases rule: rule-cases*)
case (*no-spawn p γ a p' w*)
hence $C: q \xrightarrow{p' \# w} q' \in trclAD A D \quad \forall x \in set w. x \in ssyms M \quad p' \in csyms M$
using B **by** *auto*
hence $last (p' \# w) \neq \# \wedge q' \in Q A$ **by** (*unfold sep-def*) (*auto dest: mfsm-cons' trclAD-elems*)
with C *BELOW ctrl-after-sep*[*of D q p' # w q'*] **show** ($q' \in sstates A$) **by** (*fast dest: mfsm-cons'*)
next
case (*spawn p γ a p1 w1 p2 w2*)

```

hence  $C: q \xrightarrow{p1 \# w1 @ \# p2 \# w2} q' \in \text{trclAD } A \ D \ \forall x \in \text{set } w2. x \in \text{ssyms } M \ p2 \in \text{csyms } M$  using  $B$  by auto
hence  $\text{last } (p1 \# w1 @ \# p2 \# w2) \neq \text{sep } M \wedge q' \in Q \ A$  by (auto dest: mfsm-cons' trclAD-elems)
with  $C \text{ BELOW } \text{ctrl-after-sep}[\text{of } D \ q \ p1 \# w1 @ \# p2 \# w2 \ q']$  show  $(q' \in \text{sstates } A)$  by (unfold sep-def, fast dest: mfsm-cons')
qed
qed

```

$ps\text{-}R \ M \ A$ preserves the property of being below $ps\text{-upper } M \ A$, and the transition relation becomes strictly greater in each step

lemma (in MFSM) $ps\text{-R-below}$: **assumes** $E: (D, D') \in ps\text{-R } M \ A$

shows $D \subset D' \wedge D' \subseteq ps\text{-upper } M \ A$

proof –

from E **have** $\text{BELOW}: D \subseteq ps\text{-upper } M \ A$ **by** *(simp add: ps-R-dom-below)*

{

fix $p \ \gamma \ a \ c' \ q \ q'$

assume $A: [p, \gamma] \xrightarrow{a} c' \in \text{rules } M \ q \xrightarrow{c'} q' \in \text{trclAD } A \ D$

obtain $p' \ cr' \text{ where } \text{CSPLIT}: p \in \text{csyms } M \wedge p' \in \text{csyms } M \wedge c' = p' \# cr' \wedge \gamma \in \text{ssyms } M$ **by** *(insert A) (erule rule-cases, fast+)*

with $\text{BELOW } A$ **obtain** $qh \text{ where } \text{SPLIT}: (q, p', qh) \in D \ (q, p', qh) \in ps\text{-upper } M \ A$ **by** *(fast dest: trclAD-uncons)*

with CSPLIT **have** $QC: q \in \text{cstates } A \wedge qh = sp \ A \ q \ p' \text{ by } (auto \ elim: \ ps\text{-upper-cases dest: syms-part iff add: syms-sep})$

with $\text{BELOW } A$ $\text{ctrl-rule}[\text{of } D \ p \ \gamma \ a \ c' \ q \ q']$ **have** $Q'S: q' \in \text{sstates } A$ **by** *simp*

from $QC \text{ CSPLIT}$ **have** $sp \ A \ q \ p \in \text{sstates } A$ **by** *(simp add: uniqueSp)*

with $Q'S \text{ CSPLIT}$ **have** $sp \ A \ q \ p \xrightarrow{\gamma} q' \in ps\text{-upper } M \ A$ **by** *(unfold ps-upper-def, simp)*

}

with E **show** $?thesis$ **by** *(auto elim!: ps-R.cases)*

qed

As a result of this section, we get the well-foundedness of $ps\text{-R } M \ A$, and that the transition relations that occur during the saturation algorithm stay above the initial transition relation $\delta \ A$ and below $ps\text{-upper } M \ A$

theorem (in MFSM) $ps\text{-R-wf}$: $wf \ ((ps\text{-R } M \ A)^{-1})$ **using** $ps\text{-upper-finite sat-wf}$ [**where** $\alpha = id$, **simplified**] $ps\text{-R-below}$ **by** *(blast)*

theorem (in MFSM) $ps\text{-R-above-inv}$: $is\text{-inv } (ps\text{-R } M \ A) \ (\delta \ A) \ (\lambda D. \delta \ A \subseteq D)$ **by** *(auto intro: invI elim: ps-R.cases)*

theorem (in MFSM) $ps\text{-R-below-inv}$: $is\text{-inv } (ps\text{-R } M \ A) \ (\delta \ A) \ (\lambda D. D \subseteq ps\text{-upper } M \ A)$ **by** *(rule invI) (auto simp add: initial-delta-below ps-R-below)*

We can also show that the algorithm is defined for every possible initial automata

theorem (in MFSM) $total$: $\exists D. (\delta \ A, D) \in \text{ndet-algo}(ps\text{-R } M \ A)$ **using** $ps\text{-R-wf}$ $ndet\text{-algo-total}$ **by** *blast*

4.4.2 Soundness

The soundness (over-approximation) proof works by induction over the definition of pre^* .

In the reflexive case, a configuration from the original language is also in the saturated language, because no transitions are killed during saturation.

In the step case, we assume that a configuration c' is in the saturated language, and show for a rewriting step $c \xrightarrow{a} c'$ that also c is in the saturated language.

theorem (in MFSM) sound: $\llbracket c \in pre\text{-star}(\text{rules } M) A; (\delta A, s') \in ndet\text{-algo}(ps\text{-}R M A) \rrbracket \implies c \in lang(A \parallel \delta := s')$

proof –

let $?A' = A \parallel \delta := s'$

assume $A: (\delta A, s') \in ndet\text{-algo}(ps\text{-}R M A)$

— Some little helpers

from A $ps\text{-}R$ -above-inv have $SUBSET: \delta A \subseteq s'$ by (unfold $ndet\text{-algo}\text{-def}$) (auto dest: inv)

have $TREQ: !D . trclAD A D = trclAD ?A' D$ by (rule $trclAD\text{-eq}$, simp-all)

from A $ps\text{-}R$ -below-inv have $SATSETU: \delta ?A' \subseteq ps\text{-upper } M A$ by (erule-tac $ndet\text{-algo}E$) (auto dest: inv iff add: initial-delta-below)

assume $c \in pre\text{-star}(\text{rules } M) A$

— Make an induction over the definition of pre^*

thus $?thesis$ proof (induct c rule: $pre\text{-star}.\text{induct}$)

fix c assume $c \in lang A$ — Reflexive case: The configuration comes from the original regular language

then obtain f where $F: f \in F A \wedge (s0 A, c, f) \in trclA A$ by (unfold $lang\text{-def langs-def}$, fast) — That is, c can bring the initial automata from its start state to some final state f

with $SUBSET trclAD\text{-mono-adv}[of \delta A s' A ?A']$ have $(s0 A, c, f) \in trclA ?A'$ by (auto) — Because the original transition relation δA is a subset of the saturated one s' ($SUBSET$) and the transitive closure is monotonous, $(s0 A, c, f)$ is also in the transitive closure of the saturated transition relation

with F show $c \in lang ?A'$ by (unfold $lang\text{-def langs-def}$) auto — and thus in the language of the saturated automaton

next

— Step case:

fix $a c c'$

assume $IHP: c' \in pre^*(\text{rules } M) A (c, a, c') \in tr(\text{rules } M)$ — We take some configurations c and $c' \in pre^*(\text{rules } M) A$ and assume that c can be rewritten to c' in one step

$c' \in lang ?A'$ — We further assume that c' is in the saturated language, and we have to show that also c is in that language

from IHP obtain f where $F: f \in F ?A' \wedge (s0 ?A', c', f) \in trclA ?A'$ by (unfold $lang\text{-def langs-def}$, fast) — Unfolding the definition of $lang$

from IHP obtain $w1 w2 r r'$ where $CREW: c = w1 @ (r @ w2) \wedge c' = w1 @ (r' @ w2) \wedge (r, a, r') \in \text{rules } M$ by (auto elim!: $tr.\text{cases}$) — Get the rewrite rule that rewrites

c to c'

then obtain $p \gamma p' w'$ **where** $RFMT: p \in csyms M \wedge p' \in csyms M \wedge \gamma \in ssyms M \wedge r = [p, \gamma] \wedge r' = p' \# w'$ **by** (auto elim!: rule-cases) — This rewrite rule rewrites some control symbol p followed by a stack symbol γ to another control symbol p' and a sequence of further symbols w'

with $F CREW$ **obtain** $q qh q'$ **where** $SPLIT: (s0 ?A', w1, q) \in trclA ?A' \wedge (q, p' \# w', q') \in trclA ?A' \wedge (q', w2, f) \in trclA ?A' \wedge (q, p', qh) \in \delta ?A'$

by (blast dest: $trclAD\text{-unconcat}$ $trclAD\text{-uncons}$) — Get the states in the transition relation generated by the algorithm, that correspond to the splitting of c' as established in $CREW$

have $SHORTCUT: (q, [p, \gamma], q') \in trclA ?A'$ — In the transition relation generated by our algorithm, we can get from q to q' also by $[p, \gamma]$

proof —

have $S1: (q, p, sp A q p) \in \delta ?A'$ **and** $QINC: q \in cstates A$ — The first transition, from q with p to $sp A q p$ is already contained in the initial M-automata. We also need to know for further proofs, that q is a control state.

proof —

from $SPLIT SATSETU$ **have** $(q, p', qh) \in ps\text{-upper } M A$ **by** auto

with $RFMT$ **show** $q \in cstates A$ **by** (auto elim!: $ps\text{-upper-cases}$ dest: $mfsm\text{-cons}'$ simp add: sep-def)

with $RFMT$ **have** $(q, p, sp A q p) \in \delta A$ **by** (fast intro: $cstate\text{-succ-ex}$)

with $SUBSET$ **show** $(q, p, sp A q p) \in \delta ?A'$ **by** auto

qed

moreover

have $S2: (sp A q p, \gamma, q') \in \delta ?A'$ — The second transition, from $sp A q p$ with γ to q' has been added during the algorithm's execution

proof —

from A **have** $s' \notin Domain (ps\text{-R } M A)$ **by** (blast dest: $termstate\text{-ndet-algo}$)

moreover **from** $CREW RFMT SPLIT TREQ SATSETU$ **have** $(sp A q p, \gamma, q') \notin s' \implies (s', \text{insert } (sp A q p, \gamma, q') s') \in (ps\text{-R } M A)$ **by** (auto intro: $ps\text{-R}.intros$)

ultimately show $?thesis$ **by** auto

qed

moreover

have $sp A q p \in Q ?A' \wedge q' \in Q ?A' \wedge q \in Q ?A' \wedge p \in \Sigma ?A' \wedge \gamma \in \Sigma ?A'$ — The intermediate states and labels have also the correct types

proof —

from $S2 SATSETU$ **have** $(sp A q p, \gamma, q') \in ps\text{-upper } M A$ **by** auto

with $QINC RFMT$ **show** $?thesis$ **by** (auto elim!: $ps\text{-upper-cases}$ dest: $mfsm\text{-cons}'$ simp add: states-part alpha-cons)

qed

ultimately show $?thesis$ **by** simp

qed

have $(s0 ?A', w1 @ (([p, \gamma]) @ w2), f) \in trclA ?A'$ — Now we put the pieces together and construct a path from $s0 A$ with $w1$ to q , from there with $[p, \gamma]$ to q' and then with $w2$ to the final state f

```

proof -
  from SHORTCUT SPLIT have  $(q, ([p, \gamma]) @ w2, f) \in \text{trclA } ?A'$  by (fast dest: trclAD-concat)
    with SPLIT show ?thesis by (fast dest: trclAD-concat)
    qed
    with CREW RFMT have  $(s0 ?A', c, f) \in \text{trclA } ?A'$  by auto — this is because  $c = w1 @ [p, \gamma] @ w2$ 
    with F show  $c \in \text{lang } ?A'$  by (unfold lang-def langs-def, fast) — And thus  $c$  is in the language of the saturated automaton
    qed
  qed

```

4.4.3 Precision

In this section we show the precision of the algorithm, that is we show that the saturated language is below the backwards reachable set.

The following induction scheme makes an induction over the number of occurrences of a certain transition in words accepted by a FSM:

To prove a proposition for all words from state qs to state qf in FSM A that has a transition rule $(s, a, s') \in \delta A$, we have to show the following:

- Show, that the proposition is valid for words that do not use the transition rule $(s, a, s') \in \delta A$ at all
- Assuming that there is a prefix wp from qs to s and a suffix ws from s' to qf , and that wp does not use the new rule, and further assuming that for all prefixes wh from qs to s' , the proposition holds for $wh @ ws$, show that the proposition also holds for $wp @ a \# ws$.

We actually do use D here instead of δA , for use with trclAD .

```

lemma ins-trans-induct[consumes 1, case-names base step]:
  fixes qs and qf
  assumes A:  $(qs, w, qf) \in \text{trclAD } A$  (insert  $(s, a, s') D$ )
  assumes BASE-CASE:  $\text{!! } w . (qs, w, qf) \in \text{trclAD } A \text{ } D \implies P w$ 
  assumes STEP-CASE:  $\text{!! } wp \text{ } ws . \llbracket (qs, wp, s) \in \text{trclAD } A \text{ } D; (s', ws, qf) \in \text{trclAD } A \text{ } (insert (s, a, s') D); \text{!! } wh . (qs, wh, s') \in \text{trclAD } A \text{ } D \implies P (wh @ ws) \rrbracket \implies P (wp @ a \# ws)$ 
  shows P w
proof -
  — Essentially, the proof works by induction over the suffix  $ws$ 
  {
    fix ws
    have  $\text{!! } qh \text{ } wp . \llbracket (qs, wp, qh) \in \text{trclAD } A \text{ } D; (qh, ws, qf) \in \text{trclAD } A \text{ } (insert (s, a, s') D) \rrbracket \implies P (wp @ ws)$ 
    proof (induct ws)
      case (Nil qh wp) with BASE-CASE show ?case by (subgoal-tac qh=qf, auto)
      next
      case (Cons e w qh wp) note IHP=this
  }

```

```

then obtain qhh where SPLIT:  $(qh, e, qhh) \in (insert (s \hookrightarrow_a s') D) \wedge$ 
 $(qhh, w, qf) \in trclAD A (insert (s \hookrightarrow_a s') D) \wedge qh \in Q A \wedge e \in \Sigma A$  by (fast dest:
trclAD-uncons)
show ?case proof (cases  $(qh, e, qhh) = (s, a, s')$ )
case False
with SPLIT have  $(qh, [e], qhh) \in trclAD A D$  by (auto intro: trclAD-one-elem
dest: trclAD-elems)
with IHP have  $(qs, wp@[e], qhh) \in trclAD A D$  by (fast intro: trclAD-concat)
with IHP SPLIT have  $P ((wp@[e])@w)$  by fast
thus ?thesis by simp
next
case True note CASE=this
with SPLIT IHP have  $(qs, wp, s) \in trclAD A D \wedge s' \hookrightarrow_w qf \in trclAD A$ 
 $(insert (s \hookrightarrow_a s') D) !!wh. (qs, wh, s') \in trclAD A D \implies P (wh@w)$  by simp-all
with STEP-CASE CASE show ?thesis by simp
qed
qed
} note C=this
from A C[of [] qs w] show ?thesis by (auto dest: trclAD-elems)
qed

```

The following lemma is a stronger elimination rule than *ps-R.cases*. It makes a more fine-grained distinction. In words: A step of the algorithm adds a transition $(sp A q p, \gamma, s')$, if there is a rule $([p, \gamma], a, p' \# c')$, and a transition sequence $(q, p' \# c', s') \in trclAD A D$. That is, if we have $(sp A q p', c', s') \in trclAD A D$.

```

lemma (in MFSM) ps-R-elims-adv:
assumes  $(D, D') \in ps\text{-}R M A$ 
obtains  $\gamma s' a p' c' p q$  where
 $D' = insert (sp A q p, \gamma, s') D (sp A q p, \gamma, s') \notin D [p, \gamma] \hookrightarrow_a p' \# c' \in rules M$ 
 $(q, p' \# c', s') \in trclAD A D$ 
 $p \in csyms M \gamma \in ssyms M q \in cstates A p' \in csyms M a \in labels M (q, p', sp A q p') \in D$ 
 $(sp A q p', c', s') \in trclAD A D$ 
using assms
proof (cases rule: ps-R.cases)
case A:  $(1 p \gamma a c' q q')$ 
then obtain  $p' cc'$  where RFMT:  $p \in csyms M \wedge c' = p' \# cc' \wedge p' \in csyms M \wedge$ 
 $\gamma \in ssyms M \wedge a \in labels M$  by (auto elim!: rule-cases)
with A obtain  $qh$  where SPLIT:  $(q, p', qh) \in D \wedge (qh, cc', q') \in trclAD A D$  by
(fast dest: trclAD-uncons)
with A RFMT have  $q \in cstates A \wedge qh = sp A q p'$  by (subgoal-tac  $(q, p', qh) \in ps\text{-}upper$ 
 $M A$ ) (auto elim!: ps-upper-cases dest: syms-part sep-out-syms)
then show ?thesis using A RFMT SPLIT that by blast
qed

```

Now follows a helper lemma to establish the precision result. In the original paper [1] it is called the *crucial point* of the precision proof.

It states that for transition relations that occur during the execution of the

algorithm, for each word w that leads from the start state to a state $sp A q p$, there is a word $ws @ [p]$ that leads to $sp A q p$ in the initial automaton and w can be rewritten to $ws @ [p]$.

In the initial transition relation, a state of the form $sp A q p$ has only one incoming edge labelled p (*MFSM.sp-pred-ex* *MFSM.sp-pred-unique*). Intuitively, this lemma explains why it is correct to add further incoming edges to $sp A q p$: All words using such edges can be rewritten to a word using the original edge.

lemma (in MFSM) sp-property:

shows *is-inv* (*ps-R M A*) (δA) (λD .

$(\forall w . \forall p \in \text{csyms } M. \forall q \in \text{cstates } A. (s0 A, w, sp A q p) \in \text{trclAD } A D \longrightarrow (\exists ws as. (s0 A, ws, q) \in \text{trclA } A \wedge (w, as, ws @ [p]) \in \text{trcl } (\text{tr } (\text{rules } M))) \wedge (\forall P'. \text{is-inv } (\text{ps-R } M A) (\delta A) P' \longrightarrow P' D))$

— We show the thesis by proving that it is an invariant of the saturation procedure

proof (*rule inv-useI; intro alli ballI impI conjI*)

— Base case, show the thesis for the initial automata

fix $w p q$

assume $A: p \in \text{csyms } M q \in \text{cstates } A s0 A \hookrightarrow_w sp A q p \in \text{trclA } A$

show $\exists ws as. s0 A \hookrightarrow_{ws} q \in \text{trclA } A \wedge (w, as, ws @ [p]) \in \text{trcl } (\text{tr } (\text{rules } M))$

proof (*cases w rule: rev-cases*) — Make a case distinction whether w is empty

case *Nil* — w cannot be empty, because $s0$ is a control state, and sp is a stack state, and by definition of M-automata, these cannot be equal

with A **have** $s0 A = sp A q p$ **by** (*auto*)

with A *s0-fmt uniqueSp* **have** *False* **by** (*auto dest: fsm-cons'*)

thus *?thesis ..*

next

case (*snoc ws p'*) **note** *CASE=this*

with A **obtain** qh **where** $(s0 A, ws, qh) \in \text{trclA } A \wedge (qh, [p'], sp A q p) \in \text{trclA } A \wedge (qh, p', sp A q p) \in \delta A$ **by** (*fast dest: trclAD-unconcat trclAD-single*) — Get the last state qh and symbol p' before reaching sp

moreover with A **have** $p = p' \wedge qh = q$ **by** (*blast dest: sp-pred-unique*) — This symbol is p , because the p -edge from q is the only edge to $sp A q p$ in an M-automata

moreover with *CASE* **have** $(w, [], ws @ [p]) \in \text{trcl } (\text{tr } (\text{rules } M))$ **by** (*fast intro: trcl.empty*)

ultimately show *?thesis* **by** (*blast*)

qed

next

— Step case

fix $D1 D2 w p q$

assume

IH: $\forall w. \forall p \in \text{csyms } M. \forall q \in \text{cstates } A. s0 A \hookrightarrow_w sp A q p \in \text{trclAD } A D1$

$\longrightarrow (\exists ws as. s0 A \hookrightarrow_{ws} q \in \text{trclAD } A (\delta A) \wedge (w \hookrightarrow_{as} ws @ [p]) \in \text{trcl } (\text{tr } (\text{rules } M)))$ — By induction hypothesis, our proposition is valid for $D1$

and *SUCC*: $(D1, D2) \in \text{ps-R } M A$ — We have to show the proposition for some $D2$, that is a successor state of $D1$ w.r.t. *ps-R M A*

and *P1*: $p \in \text{csyms } M q \in \text{cstates } A$ **and** *P2*: $s0 A \hookrightarrow_w sp A q p \in \text{trclAD } A D2$ — Premise of our proposition: We reach some state $sp A q p$

and *USE-INV*: $\bigwedge P'. \text{is-inv } (\text{ps-R } M A) (\delta A) P' \implies P' D1$ — We can use

known invariants

from *SUCC* **have** $SS: D1 \subseteq ps\text{-upper } M A$ **by** (*blast dest: ps-R-dom-below*)
from *USE-INV* **have** $A2: \delta A \subseteq D1$ **by** (*blast intro: ps-R-above-inv*)

from *SUCC* **obtain** $\gamma s' pp aa cc' qq$ **where** $ADD: insert(sp A qq pp, \gamma, s') D1 = D2 \wedge (sp A qq pp, \gamma, s') \notin D1$ **and**

$RCONS: ([pp, \gamma], aa, cc') \in rules M \wedge (qq, cc', s') \in trclAD A D1 \wedge qq \in cstates A \wedge pp \in csyms M \wedge aa \in labels M$

by (*blast elim!: ps-R-elims-adv*) — Because of *SUCC*, we obtain $D2$ by adding a (new) transition $(sp A qq pp, \gamma, s')$ to $D1$, such that there is a rule $([pp, \gamma], aa, cc') \in rules M$ and the former transition relation can do $(qq, cc', s') \in trclAD A D1$

from $P2$ **ADD have** $P2': s0 A \hookrightarrow_w sp A q p \in trclAD A$ ($insert(sp A qq pp \hookrightarrow_\gamma s') D1$) **by** *simp*

show $\exists ws as. s0 A \hookrightarrow_{ws} q \in trclAD A \wedge w \hookrightarrow_{as} ws @ [p] \in trcl (tr (rules M))$
using $P2'$

— We show the proposition by induction on how often the new rule was used. For this, we regard a prefix until the first usage of the added rule, and a suffix that may use the added rule arbitrarily often

proof (*induction rule: ins-trans-induct*)

case (*base*) — Base case, the added rule is not used at all. The proof is straightforward using the induction hypothesis of the outer (invariant) induction

thus $?case$ **using** *IH P1* **by** *simp*

next

fix *wpre wsfx* — Step case: We have a prefix that does not use the added rule, then a usage of the added rule and a suffix. We know that our proposition holds for all prefixes that do not use the added rule.

assume $IP1: (s0 A, wpre, sp A qq pp) \in trclAD A D1$ **and** $IP2: (s', wsfx, sp A q p) \in trclAD A$ ($insert(sp A qq pp, \gamma, s') D1$)

assume $IH: !!wh. (s0 A, wh, s') \in trclAD A D1 \implies \exists ws as. (s0 A, ws, q) \in trclAD A (\delta A) \wedge ((wh @ wsfx, as, ws @ [p]) \in trcl (tr (rules M)))$

from $IP1$ **IH RCONS obtain** *wps aps* **where** $C1: (s0 A, wps, qq) \in trclAD A (\delta A) \wedge wpre \hookrightarrow_{aps} wps @ [pp] \in trcl (tr (rules M))$ **by** *fast* — This is an instance of a configuration reaching a *sp-state*, thus by induction hypothesis of the outer (invariant) induction, we find a successor configuration *wps* @ *[pp]* that reaches this state using *pp* as last edge in δA

with $A2$ **have** $(s0 A, wps, qq) \in trclAD A D1$ **by** (*blast dest: trclAD-mono*) — And because $\delta A \subseteq D1$, we can do the transitions also in $D1$

with *RCONS* **have** $(s0 A, wps @ cc', s') \in trclAD A D1$ **by** (*blast intro: trclAD-concat*) — From above (*RCONS*) we know $(qq, cc', s') \in trclAD A D1$, and we can concatenate these transition sequences

then obtain *ws as* **where** $C2: (s0 A, ws, q) \in trclAD A (\delta A) \wedge (wps @ cc') @ wsfx \hookrightarrow_{as} ws @ [p] \in trcl (tr (rules M))$ **by** (*fast dest: IH*) — This concatenation is a prefix to a usage of the added transition, that does not use the added transition itself. (The whole configuration bringing us to $sp A q p$ is $wps @ cc' @ wsfx$). For those prefixes, we can apply the induction hypothesis of the inner induction and obtain a configuration *ws* @ *[p]* that is a successor configuration of *wps* @ *cc'* @

$wsfx$, and with which we can reach $sp A q p$ using p as last edge

have $\exists as. wpre @ \gamma \# wsfx \hookrightarrow_{as} ws @ [p] \in trcl (tr (rules M))$ — Now we obtained some configuration $ws @ [p]$, that reaches $sp A q p$ using p as last edge in δA . Now we show that this is indeed a successor configuration of $wpre @ \gamma \# wsfx$.

proof —

— This is done by putting together the transitions and using the extensibility of string rewrite systems, i.e. that we can still do a rewrite step if we add context

from $C1$ **have** $wpre @ (\gamma \# wsfx) \hookrightarrow_{aps} (wps @ [pp]) @ (\gamma \# wsfx) \in trcl (tr (rules M))$ **by** (fast intro: srs-ext)

hence $wpre @ (\gamma \# wsfx) \hookrightarrow_{aps} wps @ ([pp, \gamma]) @ wsfx \in trcl (tr (rules M))$ **by** simp

moreover from $RCONS$ **have** $wps @ ([pp, \gamma]) @ wsfx \hookrightarrow_{[aa]} wps @ cc' @ wsfx \in trcl (tr (rules M))$ **by** (fast intro: tr.rewrite trcl-one-elem)

hence $wps @ ([pp, \gamma]) @ wsfx \hookrightarrow_{[aa]} (wps @ cc') @ wsfx \in trcl (tr (rules M))$ **by** simp

moreover note $C2$

ultimately have $wpre @ (\gamma \# wsfx) \hookrightarrow_{aps} wps @ [aa] @ as ws @ [p] \in trcl (tr (rules M))$ **by** (fast intro: trcl-concat)

thus ?thesis **by** fast

qed

with $C2$ **show** $\exists ws as. s0 A \hookrightarrow_{ws} q \in trcl A A \wedge wpre @ \gamma \# wsfx \hookrightarrow_{as} ws @ [p] \in trcl (tr (rules M))$ **by** fast — Finally, we have the proposition for the configuration $wpre @ \gamma \# wsfx$, that contains the added rule (s, γ, s') one time more

qed

qed

Helper lemma to clarify some subgoal in the precision proof:

lemma $trclAD\text{-delta-update-inv}$: $trclAD (A(\delta := X)) D = trclAD A D$ **by** (simp add: $trclAD$ -by- $trcl'$)

The precision is proved as an invariant of the saturation algorithm:

theorem (in MFSM) precise-inv:

shows $is\text{-inv} (ps\text{-}R M A) (\delta A) (\lambda D. (lang (A(\delta := D))) \subseteq pre^* (rules M) A) \wedge (\forall P'. is\text{-inv} (ps\text{-}R M A) (\delta A) P' \longrightarrow P' D)$

proof —

{

fix $D1 D2 w f$

assume $IH: \{w. \exists f \in F A. s0 A \hookrightarrow_w f \in trclAD A D1\} \subseteq pre^* (rules M) A$ — By induction hypothesis, we know $lang (A(\delta := D1)) \subseteq pre^* (rules M) A$

assume $SUCC: (D1, D2) \in ps\text{-}R M A$ — We regard a successor $D2$ of $D1$ w.r.t. $ps\text{-}R M A$

assume $P1: f \in F A$ **and** $P2: s0 A \hookrightarrow_w f \in trclAD A D2$ — And a word $w \in lang (A(\delta := D2))$

assume $USE\text{-INV}$: $\bigwedge P'. is\text{-inv} (ps\text{-}R M A) (\delta A) P' \implies P' D1$ — For the proof, we can use any known invariants

from *SUCC obtain* $\gamma s' p a c' q$ **where** *ADD: insert* $(sp A q p, \gamma, s') D1 = D2$
 $\wedge (sp A q p, \gamma, s') \notin D1$ **and**
RCONS: $([p, \gamma], a, c') \in \text{rules } M \wedge (q, c', s') \in \text{trclAD } A D1 \wedge q \in \text{cstates } A \wedge p \in \text{csyms } M \wedge a \in \text{labels } M \wedge \gamma \in \text{ssyms } M$
by *(blast elim!: ps-R-elims-adv)* — Because of $(D1, D2) \in \text{ps-R } M A$, we obtain $D2$ by adding a (new) transition $(sp A q p, \gamma, s')$ to $D1$, such that there is a rule $([p, \gamma], a, c')$ and we have $(q, c', s') \in \text{trclAD } A D1$
from *P2 ADD have* $P2': s0 A \xrightarrow{w} f \in \text{trclAD } A (\text{insert } (sp A q p \xrightarrow{\gamma} s') D1)$
by *simp*
from *SUCC have* *SS: $D1 \subseteq \text{ps-upper } M A$* **by** *(blast dest: ps-R-dom-below)* —
 We know, that the intermediate value is below the upper saturation bound
from *USE-INV have* *A2: $\delta A \subseteq D1$* **by** *(blast intro: ps-R-above-inv)* — ... and above the start value
from *SS USE-INV sp-property have* *SP-PROP: $(\forall w. \forall p \in \text{csyms } M. \forall q \in \text{cstates } M. (s0 A, w, sp A q p) \in \text{trclAD } A D1 \longrightarrow (\exists ws as. (s0 A, ws, q) \in \text{trclA } A \wedge (w, as, ws @ [p]) \in \text{trcl } (\text{tr } (\text{rules } M)))$*
by *blast* — And we have just shown *sp-property*, that tells us that each configuration w that leads to a state $sp A q p$, can be rewritten to a configuration in the initial automaton, that uses p as its last transition

have $w \in \text{pre}^* (\text{rules } M) A$ **using** *P2'* — We have to show that the word w from the new automaton is also in $\text{pre}^* (\text{rules } M) A$. We show this by induction on how often the new transition is used by w
proof (rule ins-trans-induct)
fix wa **assume** $(s0 A, wa, f) \in \text{trclAD } A D1$ — Base case: w does not use the new transition at all
with *IH P1 show* $wa \in \text{pre}^* (\text{rules } M) A$ **by** *(fast)* — The proposition follows directly from the outer (invariant) induction and can be solved automatically
next
fix $wpre wsfx$ — Step case
assume *IP1: $(s0 A, wpre, sp A q p) \in \text{trclAD } A D1$* — We assume that we have a prefix $wpre$ leading to the start state s of the new transition and not using the new transition
assume *IP2: $(s', wsfx, f) \in \text{trclAD } A (\text{insert } (sp A q p, \gamma, s') D1)$* — We also have a suffix from the end state s' to f
assume *IIH: $!!wh. (s0 A, wh, s') \in \text{trclAD } A D1 \implies wh @ wsfx \in \text{pre}^* (\text{rules } M) A$* — And we assume that our proposition is valid for prefixes wh that do not use the new transition
 — We have to show that the proposition is valid for $wpre @ \gamma \# wsfx$
from *IP1 SP-PROP RCONS obtain* $wpres apres$ **where** *SPP: $(s0 A, wpres, q) \in \text{trclA } A \wedge wpre \xrightarrow{\gamma} apres$* $wpres @ [p] \in \text{trcl } (\text{tr } (\text{rules } M))$ **by** *(blast)* — We can apply *SP-PROP*, to find a successor $wpres @ [p]$ of $wpre$ in the initial automata
with *A2 have* $s0 A \xrightarrow{wpres} q \in \text{trclAD } A D1$ **by** *(blast dest: trclAD-mono)*
 — $wpres$ can also be read by $D1$ because of $\delta A \subseteq D1$
with *RCONS have* $s0 A \xrightarrow{wpres @ c'} s' \in \text{trclAD } A D1$ **by** *(fast intro: trclAD-concat)* — Altogether we get a prefix $wpres @ c'$ that leads to s' , without using the added transition
with *IIH have* $(wpres @ c') @ wsfx \in \text{pre-star } (\text{rules } M) A$ **by** *fast* — We can apply the induction hypothesis

then obtain as wo **where** $C1: wpres@c'@wsfx \hookrightarrow_{as} wo \in trcl(tr(rules M)) \wedge wo \in lang A$ **by** (auto elim!: pre-star-altE) — And find that there is a wo in the original automata, that is a successor of $wpres @ c' @ wsfx$

moreover have $\exists as. wpre@\gamma\#wsfx \hookrightarrow_{as} wo \in trcl(tr(rules M))$ — Next we show that wo is a successor of $wpre @ \gamma \# wsfx$

proof —

from SPP **have** $wpre@\gamma\#wsfx \hookrightarrow_{apres} (wpres@[p])@\gamma\#wsfx \in trcl(tr(rules M))$ **by** (fast intro: srs-ext)

hence $wpre@\gamma\#wsfx \hookrightarrow_{apres} wpres@[p, \gamma]@\gamma\#wsfx \in trcl(tr(rules M))$ **by** simp

moreover from $RCONS$ **have** $wpres@[p, \gamma]@\gamma\#wsfx \hookrightarrow_{[a]} wpres@c'@wsfx \in trcl(tr(rules M))$ **by** (fast intro: tr.rewrite trcl-one-elem)

moreover note $C1$

ultimately show $?thesis$ **by** (fast intro: trcl-concat)

qed

ultimately show $wpre @ \gamma \# wsfx \in pre^*(rules M) A$ **by** (fast intro: pre-star-altI) — And altogether we have $wpre @ \gamma \# wsfx \in pre^*(rules M) A$

qed

} note $A = this$

show $?thesis$

apply (rule inv-useI)

subgoal **by** (auto intro: pre-refl) — The base case is solved automatically, it follows from the reflexivity of pre^* .

subgoal for $D s'$

unfolding lang-def langs-def

using A **by** (fastforce simp add: trclAD-delta-update-inv)

done

qed

As precision is an invariant of the saturation algorithm, and is trivial for the case of an already saturated initial automata, the result of the saturation algorithm is precise

corollary (in $MFSM$) **precise**: $\llbracket (\delta A, D) \in ndet-algo(ps-R M A); x \in lang(A(\delta := D)) \rrbracket \implies x \in pre-star(rules M) A$

by (auto elim!: ndet-algoE dest: inv intro: precise-inv pre-refl)

And finally we get correctness of the algorithm, with no restrictions on valid states

theorem (in $MFSM$) **correct**: $\llbracket (\delta A, D) \in ndet-algo(ps-R M A) \rrbracket \implies lang(A(\delta := D)) = pre-star(rules M) A$ **by** (auto intro: precise sound)

So the main results of this theory are, that the algorithm is defined for every possible initial automata

$MFSM ?M ?A \implies \exists D. (\delta ?A, D) \in ndet-algo(ps-R ?M ?A)$

and returns the correct result

$\llbracket MFSM ?M ?A; (\delta ?A, ?D) \in ndet-algo(ps-R ?M ?A) \rrbracket \implies lang(?A(\delta := ?D)) = pre^*(rules ?M) ?A$

We could also prove determination, i.e. the terminating state is uniquely determined by the initial state (though there may be many ways to get there). This is not really needed here, because for correctness, we do not look at the structure of the final automaton, but just at its language. The language of the final automaton is determined, as implied by *MFSM.correct*.
end

5 Non-executable implementation of the DPN pre*-algorithm

```
theory DPN-impl
imports DPN
begin
```

This theory is to explore how to prove the correctness of straightforward implementations of the DPN pre* algorithm. It does not provide an executable specification, but uses set-datatype and the SOME-operator to describe a deterministic refinement of the nondeterministic pre*-algorithm. This refinement is then characterized as a recursive function, using recdef.

This proof uses the same techniques to get the recursive function and prove its correctness as are used for the straightforward executable implementation in DPN_implEx. Differences from the executable specification are:

- The state of the algorithm contains the transition relation that is saturated, thus making the refinement abstraction just a projection onto this component. The executable specification, however, uses list representation of sets, thus making the refinement abstraction more complex.
- The termination proof is easier: In this approach, we only do recursion if our state contains a valid M-automata and a consistent transition relation. Using this property, we can infer termination easily from the termination of *ps-R*. The executable implementation does not check whether the state is valid, and thus may also do recursion for invalid states. Thus, the termination argument must also regard those invalid states, and hence must be more general.

5.1 Definitions

```
type-synonym ('c,'l,'s,'m1,'m2) pss-state = (((('c,'l,'m1) DPN-rec-scheme * ('s,'c,'m2)
MFSM-rec-scheme) * ('s,'c) LTS)
```

Function to select next transition to be added

```
definition pss-isNext :: ('c,'l,'m1) DPN-rec-scheme ⇒ ('s,'c,'m2) MFSM-rec-scheme
⇒ ('s,'c) LTS ⇒ ('s*'c*'s) ⇒ bool where
```

$pss\text{-}isNext M A D t == t \notin D \wedge (\exists q p \gamma q' a c'. t = (sp A q p, \gamma, q') \wedge [p, \gamma] \hookrightarrow_a c' \in \text{rules } M \wedge (q, c', q') \in \text{trclAD } A D)$
definition $pss\text{-}next M A D == \text{if } (\exists t. pss\text{-}isNext } M A D t) \text{ then Some } (\text{SOME } t. pss\text{-}isNext } M A D t) \text{ else None}$

Next state selector function

definition

$pss\text{-}next\text{-}state } S == \text{case } S \text{ of } ((M, A), D) \Rightarrow \text{if } \text{MFSM } M A \wedge D \subseteq \text{ps-upper } M A \text{ then } (\text{case } pss\text{-}next M A D \text{ of None } \Rightarrow \text{None} \mid \text{Some } t \Rightarrow \text{Some } ((M, A), \text{insert } t D)) \text{ else None}$

Relation describing the deterministic algorithm

definition

$pss\text{-}R == \text{graph } pss\text{-}next\text{-}state$

lemma $pss\text{-}nextE1: pss\text{-}next M A D = \text{Some } t \implies t \notin D \wedge (\exists q p \gamma q' a c'. t = (sp A q p, \gamma, q') \wedge [p, \gamma] \hookrightarrow_a c' \in \text{rules } M \wedge (q, c', q') \in \text{trclAD } A D)$

proof –

assume $pss\text{-}next M A D = \text{Some } t$

hence $pss\text{-}isNext M A D t$

apply (*unfold pss-next-def*)

apply (*cases* $\exists t. pss\text{-}isNext M A D t$)

by (*auto intro: someI*)

thus *?thesis* **by** (*unfold pss-isNext-def*)

qed

lemma $pss\text{-}nextE2: pss\text{-}next M A D = \text{None} \implies \neg(\exists q p \gamma q' a c'. t \notin D \wedge t = (sp A q p, \gamma, q') \wedge [p, \gamma] \hookrightarrow_a c' \in \text{rules } M \wedge (q, c', q') \in \text{trclAD } A D)$

proof –

assume $pss\text{-}next M A D = \text{None}$

hence $\neg(\exists t. pss\text{-}isNext M A D t)$

apply (*unfold pss-next-def*)

apply (*cases* $\exists t. pss\text{-}isNext M A D t$)

by *auto*

thus *?thesis* **by** (*unfold pss-isNext-def*) *blast*

qed

lemmas (in MFSM) $pss\text{-}nextE = pss\text{-}nextE1 \ pss\text{-}nextE2$

The relation of the deterministic algorithm is also the recursion relation of the recursive characterization of the algorithm

lemma $pss\text{-}R\text{-}alt[\text{termination-simp}]: pss\text{-}R == \{((M, A), D), ((M, A), \text{insert } t D) \mid M A D t. \text{MFSM } M A \wedge D \subseteq \text{ps-upper } M A \wedge pss\text{-}next M A D = \text{Some } t\}$

by (*rule eq-reflection, unfold pss-R-def graph-def pss-next-state-def*) (*auto split: option.split-asm if-splits*)

5.2 Refining $ps\text{-}R$

We first show that the next-step relation refines $ps\text{-}R M A$. From this, we will get both termination and correctness

Abstraction relation to project on the second component of a tuple, with fixed first component

```

definition  $\alpha\text{snd } f == \{ (s, (f, s)) \mid s. \text{True} \}$ 
lemma  $\alpha\text{snd-comp-simp}: R O \alpha\text{snd } f = \{ (s, (f, s')) \mid s s'. (s, s') \in R \}$  by (unfold  $\alpha\text{snd-def}$ , blast)

lemma  $\alpha\text{sndI}[simp]: (s, (f, s)) \in \alpha\text{snd } f$  by (unfold  $\alpha\text{snd-def}$ , auto)
lemma  $\alpha\text{sndE}: (s, (f, s')) \in \alpha\text{snd } f' \implies f = f' \wedge s = s'$  by (unfold  $\alpha\text{snd-def}$ , auto)

```

Relation of $pss\text{-}next$ and $ps\text{-}R M A$

```

lemma (in MFSM)  $pss\text{-cons1}: \llbracket pss\text{-next } M A D = \text{Some } t; D \subseteq ps\text{-upper } M A \rrbracket \implies (D, \text{insert } t D) \in ps\text{-R } M A$  by (auto dest:  $pss\text{-nextE}$  intro:  $ps\text{-R.intros}$ )
lemma (in MFSM)  $pss\text{-cons2}: pss\text{-next } M A D = \text{None} \implies D \notin \text{Domain } (ps\text{-R } M A)$  by (blast dest:  $pss\text{-nextE}$  elim:  $ps\text{-R.cases}$ )

lemma (in MFSM)  $pss\text{-cons1-rev}: \llbracket D \subseteq ps\text{-upper } M A; D \notin \text{Domain } (ps\text{-R } M A) \rrbracket \implies pss\text{-next } M A D = \text{None}$  by (cases  $pss\text{-next } M A D$ ) (auto iff add:  $pss\text{-cons1}$   $pss\text{-cons2}$ )
lemma (in MFSM)  $pss\text{-cons2-rev}: \llbracket D \in \text{Domain } (ps\text{-R } M A) \rrbracket \implies \exists t. pss\text{-next } M A D = \text{Some } t \wedge (D, \text{insert } t D) \in ps\text{-R } M A$ 
by (cases  $pss\text{-next } M A D$ ) (auto iff add:  $pss\text{-cons1}$   $pss\text{-cons2}$   $ps\text{-R-dom-below}$ )

```

The refinement result

```

theorem (in MFSM)  $pss\text{-refines}: pss\text{-R} \leq_{\alpha\text{snd}} (M, A) \text{ (ps-R } M A)$  proof (rule  $\text{refinesI}$ )
  show  $\alpha\text{snd } (M, A) O pss\text{-R} \subseteq ps\text{-R } M A O \alpha\text{snd } (M, A)$  by (rule  $\text{refines-compI}$ , unfold  $\alpha\text{snd-def}$   $pss\text{-R-alt}$ ) (blast intro:  $pss\text{-cons1}$ )
next
  show  $\alpha\text{snd } (M, A) \text{ ``Domain } (ps\text{-R } M A) \subseteq \text{Domain } pss\text{-R}$ 
  apply (rule  $\text{refines-domI}$ )
  unfolding  $\alpha\text{snd-def}$   $pss\text{-R-alt}$   $\text{Domain-iff}$ 
  apply (clarify, safe)
  subgoal by unfold-locales
  subgoal by (blast dest:  $ps\text{-R-dom-below}$ )
  subgoal by (insert  $pss\text{-cons2-rev}$ , fast)
  done
qed

```

5.3 Termination

We can infer termination directly from the well-foundedness of $ps\text{-R}$ and $MFSM.pss\text{-refines}$

```
theorem  $pss\text{-R-wf}: wf (pss\text{-R}^{-1})$ 
```

```

proof -
  {
    fix  $M A D M' A' D'$ 
    assume  $A: (((M,A),D),((M',A'),D')) \in pss\text{-}R$ 
    then interpret  $MFSM \text{ sep } M M A$ 
      apply (unfold  $pss\text{-}R\text{-}alt$   $MFSM\text{-}def$ )
      apply blast
      apply simp
      done
    from  $pss\text{-}refines$   $ps\text{-}R\text{-}wf$  have  $pss\text{-}R \leq_{\alpha snd} (M, A) \text{ ps-}R M A \wedge wf ((ps\text{-}R M A)^{-1})$  by simp
  } note  $A = this$ 
  show ?thesis
    apply (rule  $\text{refines-}wf$  [ of  $pss\text{-}R$   $\text{snd } \lambda r. \alpha snd (\text{fst } r) \lambda r. \text{let } (M,A) = \text{fst } r \text{ in } ps\text{-}R M A$ ])
    using  $A$ 
    by fastforce
  
```

qed

5.4 Recursive characterization

Having proved termination, we can characterize our algorithm as a recursive function

```

function  $pss\text{-}algo\text{-}rec :: (('c,'l,'s,'m1,'m2) pss\text{-}state) \Rightarrow (('c,'l,'s,'m1,'m2) pss\text{-}state)$ 
where
   $pss\text{-}algo\text{-}rec ((M,A),D) = (\text{if } (MFSM M A \wedge D \subseteq ps\text{-}upper M A) \text{ then } (\text{case } (pss\text{-}next M A D) \text{ of } None \Rightarrow ((M,A),D) \mid (\text{Some } t) \Rightarrow pss\text{-}algo\text{-}rec ((M,A),\text{insert } t D)) \text{ else } ((M,A),D))$ 
  by pat-completeness auto

```

termination

```

apply (relation  $pss\text{-}R^{-1}$ )
apply (simp add:  $pss\text{-}R\text{-}wf$ )
using  $pss\text{-}R\text{-}alt$  by fastforce

```

```

lemma  $pss\text{-}algo\text{-}rec\text{-}newsimps[simp]:$ 
   $\llbracket MFSM M A; D \subseteq ps\text{-}upper M A; pss\text{-}next M A D = None \rrbracket \implies pss\text{-}algo\text{-}rec ((M,A),D) = ((M,A),D)$ 
   $\llbracket MFSM M A; D \subseteq ps\text{-}upper M A; pss\text{-}next M A D = Some t \rrbracket \implies pss\text{-}algo\text{-}rec ((M,A),D) = pss\text{-}algo\text{-}rec ((M,A),\text{insert } t D)$ 
   $\neg MFSM M A \implies pss\text{-}algo\text{-}rec ((M,A),D) = ((M,A),D)$ 
   $\neg (D \subseteq ps\text{-}upper M A) \implies pss\text{-}algo\text{-}rec ((M,A),D) = ((M,A),D)$ 
by auto

```

```

declare  $pss\text{-}algo\text{-}rec.simps[simp del]$ 

```

5.5 Correctness

The correctness of the recursive version of our algorithm can be inferred using the results from the locale *detRef-impl*

```

interpretation det-impl: detRef-impl pss-algo-rec pss-next-state pss-R
  apply (rule detRef-impl.intro)
  apply (simp-all add: detRef-wf-transfer[OF pss-R-wf] pss-R-def)
  subgoal for s s'
    unfolding pss-next-state-def
    by (auto split: if-splits prod.splits option.splits)
  subgoal for s
    apply (unfold pss-next-state-def)
    apply (clar simp split: prod.splits if-splits option.splits)
    using pss-algo-rec-newsimps(3,4) by blast
  done

theorem (in MFSM) pss-correct: lang (A(| δ:=snd (pss-algo-rec ((M,A), (δ A)))|)) = pre-star (rules M) A
proof –
  have (((M,A), δ A), pss-algo-rec ((M,A), δ A)) ∈ ndet-algo pss-R by (rule det-impl.algo-correct)
  moreover have (δ A, ((M,A), δ A)) ∈ αsnd (M,A) by simp
  ultimately obtain D' where 1: (D', pss-algo-rec ((M,A), δ A)) ∈ αsnd (M,A)
  and (δ A, D') ∈ ndet-algo (ps-R M A) using pss-refines by (blast dest: refines-ndet-algo)
  with correct have lang (A(| δ := D'|)) = pre* (rules M) A by auto
  moreover from 1 have snd (pss-algo-rec ((M,A), δ A)) = D' by (unfold
  αsnd-def, auto)
  ultimately show ?thesis by auto
qed

end

```

6 Tools for executable specifications

```

theory ImplHelper
imports Main
begin

```

6.1 Searching in Lists

Given a function *f* and a list *l*, return the result of the first element *e* ∈ *set l* with *f e* ≠ *None*. The functional code snippet *first-that f l* corresponds to the imperative code snippet: *for e in l do { if f e ≠ None then return Some (f e) }; return None*

```

primrec first-that :: ('s ⇒ 'a option) ⇒ 's list ⇒ 'a option where
  first-that f [] = None
  | first-that f (e#w) = (case f e of None ⇒ first-that f w | Some a ⇒ Some a)

```

```

lemma first-thatE1: first-that f l = Some a ==> ∃ e∈set l. f e = Some a
  apply (induct l)
  subgoal by simp
  subgoal for aa l by (cases f aa) auto
  done

lemma first-thatE2: first-that f l = None ==> ∀ e∈set l. f e = None
  apply (induct l)
  subgoal by simp
  subgoal for aa l by (cases f aa) auto
  done

lemmas first-thatE = first-thatE1 first-thatE2

lemma first-thatI1: e∈set l ∧ f e = Some a ==> ∃ a'. first-that f l = Some a'
  by (cases first-that f l) (auto dest: first-thatE2)

lemma first-thatI2: ∀ e∈set l. f e = None ==> first-that f l = None
  by (cases first-that f l) (auto dest: first-thatE1)

lemmas first-thatI = first-thatI1 first-thatI2

end

```

7 Executable algorithms for finite state machines

```

theory FSM-ex
imports FSM ImplHelper
begin

```

The transition relation of a finite state machine is represented as a list of labeled edges

```
type-synonym ('s,'a) delta = ('s × 'a × 's) list
```

7.1 Word lookup operation

Operation that finds some state q' that is reachable from state q with word w and has additional property P .

```

primrec lookup :: ('s ⇒ bool) ⇒ ('s,'a) delta ⇒ 's ⇒ 'a list ⇒ 's option where
  lookup P d q [] = (if P q then Some q else None)
  | lookup P d q (e#w) = first-that (λt. let (qs,es,q')=t in if q=qs ∧ e=es then lookup
    P d q' w else None) d

```

```

lemma lookupE1: !!q. lookup P d q w = Some q' ==> P q' ∧ (q,w,q') ∈ trel (set d)
proof (induct w)
  case Nil thus ?case by (cases P q) simp-all
  next
  case (Cons e w) note IHP=this

```

```

hence first-that ( $\lambda t. \text{let } (qs, es, qh) = t \text{ in if } q = qs \wedge e = es \text{ then } \text{lookup } P d qh w \text{ else } None$ )  $d = \text{Some } q'$  by simp
  then obtain  $t$  where  $t \in \text{set } d \wedge ((\text{let } (qs, es, qh) = t \text{ in if } q = qs \wedge e = es \text{ then } \text{lookup } P d qh w \text{ else } None) = \text{Some } q')$  by (blast dest: first-thatE1)
  then obtain  $qh$  where 1:  $(q, e, qh) \in \text{set } d \wedge \text{lookup } P d qh w = \text{Some } q'$ 
    by (auto split: prod.splits if-splits)
  moreover from 1 IHP have  $P q' \wedge (qh, w, q') \in \text{trcl } (\text{set } d)$  by auto
  ultimately show ?case by auto
qed

lemma lookupE2: !!q.  $\text{lookup } P d q w = None \implies \neg(\exists q'. (P q') \wedge (q, w, q') \in \text{trcl } (\text{set } d))$ 
  proof (induct w)
    case Nil thus ?case by (cases P q) (auto dest: trcl-empty-cons)
  next
    case (Cons e w) note IHP=this
    hence first-that ( $\lambda t. \text{let } (qs, es, qh) = t \text{ in if } q = qs \wedge e = es \text{ then } \text{lookup } P d qh w \text{ else } None$ )  $d = None$  by simp
    hence  $\forall t \in \text{set } d. (\text{let } (qs, es, qh) = t \text{ in if } q = qs \wedge e = es \text{ then } \text{lookup } P d qh w \text{ else } None) = None$  by (blast dest: first-thatE2)
    hence 1: !!qs es qh.  $(qs, es, qh) \in \text{set } d \implies q \neq qs \vee e \neq es \vee \text{lookup } P d qh w = None$  by auto
    show ?case proof (rule notI, elim exE conjE)
      fix  $q'$ 
      assume C:  $P q' (q, e \# w, q') \in \text{trcl } (\text{set } d)$ 
      then obtain  $qh$  where 2:  $(q, e, qh) \in \text{set } d \wedge (qh, w, q') \in \text{trcl } (\text{set } d)$  by (blast dest: trcl-uncons)
        with 1 have  $\text{lookup } P d qh w = None$  by auto
        with C 2 IHP show False by auto
      qed
    qed

lemma lookupI1:  $\llbracket P q'; (q, w, q') \in \text{trcl } (\text{set } d) \rrbracket \implies \exists q'. \text{lookup } P d q w = \text{Some } q'$ 
  by (cases lookup P d q w) (auto dest: lookupE2)

lemma lookupI2:  $\neg(\exists q'. P q' \wedge (q, w, q') \in \text{trcl } (\text{set } d)) \implies \text{lookup } P d q w = None$ 
  by (cases lookup P d q w) (auto dest: lookupE1)

lemmas lookupE = lookupE1 lookupE2
lemmas lookupI = lookupI1 lookupI2

lemma lookup-trclAD-E1:
  assumes map:  $\text{set } d = D$  and start:  $q \in Q A$  and cons:  $D \subseteq Q A \times \Sigma A \times Q A$ 
  assumes A:  $\text{lookup } P d q w = \text{Some } q'$ 
  shows  $P q' \wedge (q, w, q') \in \text{trclAD } A D$ 
proof -
  from A map have 1:  $P q' \wedge (q, w, q') \in \text{trcl } D$  by (blast dest: lookupE1)
  hence  $(q, w, q') \in \text{trcl } (D \cap (Q A \times \Sigma A \times Q A)) \cap (Q A \times \text{UNIV} \times \text{UNIV})$ 
  using cons start by (subgoal-tac D = D  $\cap (Q A \times \Sigma A \times Q A)$ , auto)

```

```

with 1 trclAD-by-trcl' show ?thesis by auto
qed

lemma lookup-trclAD-E2:
  assumes map: set d = D
  assumes A: lookup P d q w = None
  shows  $\neg (\exists q'. P q' \wedge (q, w, q') \in \text{trclAD } A \ D)$ 
  proof –
    from map A have  $\neg (\exists q'. P q' \wedge (q, w, q') \in \text{trcl } D)$  by (blast dest: lookupE2)
    with trclAD-subset-trcl show ?thesis by auto
  qed

lemma lookup-trclAD-I1:  $\llbracket \text{set } d = D; (q, w, q') \in \text{trclAD } A \ D; P q' \rrbracket \implies \exists q'. \text{lookup}$ 
  P d q w = Some q'
  apply (cases lookup P d q w)
  apply (subgoal-tac  $\neg (\exists q'. P q' \wedge (q, w, q') \in \text{trclAD } A \ D)$ )
  apply simp
  apply (rule lookup-trclAD-E2)
  apply auto
  done

lemma lookup-trclAD-I2:  $\llbracket \text{set } d = D; q \in Q \ A; D \subseteq Q \ A \times \Sigma \ A \times Q \ A; \neg (\exists q'. P$ 
  q' \wedge (q, w, q') \in \text{trclAD } A \ D) \rrbracket \implies \text{lookup } P \ d \ q \ w = \text{None}
  apply (cases lookup P d q w, auto)
  apply (subgoal-tac  $P a \wedge (q, w, a) \in \text{trclAD } A \ (\text{set } d)$ )
  apply blast
  apply (rule lookup-trclAD-E1)
  apply auto
  done

lemmas lookup-trclAD-E = lookup-trclAD-E1 lookup-trclAD-E2
lemmas lookup-trclAD-I = lookup-trclAD-I1 lookup-trclAD-I2

```

7.2 Reachable states and alphabet inferred from transition relation

```

definition states d == fst ‘ (set d) ∪ (snd ∘ snd) ‘ (set d)
definition alpha d == (fst ∘ snd) ‘ (set d)

lemma statesAlphaI:  $(q, a, q') \in \text{set } d \implies q \in \text{states } d \wedge q' \in \text{states } d \wedge a \in \text{alpha } d$  by
  (unfold states-def alpha-def, force)
lemma statesE:  $q \in \text{states } d \implies \exists a \ q'. ((q, a, q') \in \text{set } d \vee (q', a, q) \in \text{set } d)$  by (unfold
  states-def alpha-def, force)
lemma alphaE:  $a \in \text{alpha } d \implies \exists q \ q'. (q, a, q') \in \text{set } d$  by (unfold states-def alpha-def,
force)

lemma states-finite: finite (states d) by (unfold states-def, auto)
lemma alpha-finite: finite (alpha d) by (unfold alpha-def, auto)

```

```

lemma statesAlpha-subset: set d ⊆ states d × alpha d × states d by (auto dest: statesAlphaI)

lemma states-mono: set d ⊆ set d'  $\implies$  states d ⊆ states d' by (unfold states-def, auto)
lemma alpha-mono: set d ⊆ set d'  $\implies$  alpha d ⊆ alpha d' by (unfold alpha-def, auto)

lemma statesAlpha-insert: set d' = insert (q,a,q') (set d)  $\implies$  states d' = states d
 $\cup$  {q,q'}  $\wedge$  alpha d' = insert a (alpha d)
by (unfold states-def alpha-def) (simp, blast)

lemma statesAlpha-inv:  $\llbracket q \in \text{states } d; a \in \text{alpha } d; q' \in \text{states } d; \text{set } d' = \text{insert } (q, a, q') (set d) \rrbracket \implies \text{states } d = \text{states } d' \wedge \text{alpha } d = \text{alpha } d'$ 
by (unfold states-def alpha-def) (simp, blast)

export-code lookup checking SML

```

end

8 Implementation of DPN pre*-algorithm

```

theory DPN-implEx
imports DPN FSM-ex
begin

```

In this section, we provide a straightforward executable specification of the DPN-algorithm. It has a polynomial complexity, but is far from having optimal complexity.

8.1 Representation of DPN and M-automata

```

type-synonym 'c rule-ex = 'c × 'c × 'c × 'c list
type-synonym 'c DPN-ex = 'c rule-ex list

```

```

definition rule-repr == { ((p,γ,p',c'),(p#[γ],a,p'#[c'])) | p γ p' c' a . True }
definition rules-repr == { (l,l') . rule-repr " set l = l' }

```

```

lemma rules-repr-cons:  $\llbracket (R,S) \in \text{rules-repr} \rrbracket \implies ((p, \gamma, p', c') \in \text{set } R) = (\exists a. (p#[\gamma] \rightarrow_a p'#[c']) \in S)$ 
by (unfold rules-repr-def rule-repr-def) blast

```

We define the mapping to sp-states explicitly, well-knowing that it makes the algorithm even more inefficient

```

definition find-sp d s p == first-that (λt. let (sh,ph,qh)=t in if s=sh ∧ p=ph then Some qh else None) d

```

This locale describes an M-automata together with its representation used in the implementation

```
locale MFSM-ex = MFSM +
  fixes R and D
  assumes rules-repr: (R,rules M) ∈ rules-repr
  assumes D-above: δ A ⊆ set D and D-below: set D ⊆ ps-upper M A
```

This lemma exports the additional conditions of locale MFSM_ex to locale MFSM

```
lemma (in MFSM) MFSM-ex-alt: MFSM-ex M A R D ↔ (R,rules M) ∈ rules-repr
  ∧ δ A ⊆ set D ∧ set D ⊆ ps-upper M A
  using MFSM-axioms by (unfold MFSM-def MFSM-ex-def MFSM-ex-axioms-def)
  (auto)
```

```
lemmas (in MFSM-ex) D-between = D-above D-below
```

The representation of the sp-states behaves as expected

```
lemma (in MFSM-ex) find-sp-cons:
  assumes A: s ∈ cstates A p ∈ csyms M
  shows find-sp D s p = Some (sp A s p)
proof -
  let ?f = (λt. let (sh,ph,qh)=t in if s=sh ∧ p=ph then Some qh else None)
  from A have (s,p,sp A s p) ∈ set D using cstate-succ-ex' D-between by simp
  moreover have ?f (s,p,sp A s p) = Some (sp A s p) by auto
  ultimately obtain sp' where G: find-sp D s p = Some sp'
    using first-thatI1[of (s,p,sp A s p) D ?f sp A s p] by (unfold find-sp-def, blast)
    with first-thatE1[?f D sp'] obtain t where t ∈ set D ∧ ?f t = Some sp' by
    (unfold find-sp-def, blast)
    hence (s,p,sp') ∈ set D by (cases t, auto split: if-splits)
    with A D-between have sp' = sp A s p using cstate-succ-unique' by simp
    with G show ?thesis by simp
qed
```

8.2 Next-element selection

The implementation goes straightforward by implementing a function to return the next transition to be added to the transition relation of the automata being saturated

```
definition sel-next:: 'c DPN-ex ⇒ ('s,'c) delta ⇒ ('s × 'c × 's) option where
  sel-next R D ==
    first-that (λr. let (p,γ,p',c') = r in
      first-that (λt. let (q,pp',sp') = t in
        if pp'=p' then
          case find-sp D q p of
            Some spt ⇒ (case lookup (λq'. (spt,γ,q') ∉ set D) D sp' c' of
              Some q' ⇒ Some (spt,γ,q') |
              None ⇒ None
```

```

) | - ⇒ None
else None
) D
) R

```

The state of our algorithm consists of a representation of the DPN-rules and a representation of the transition relations of the automata being saturated
type-synonym $('c,'s) \text{ seln-state} = 'c \text{ DPN-ex} \times ('s,'c) \text{ delta}$

As long as the next-element function returns elements, these are added to the transition relation and the algorithm is applied recursively. *sel-next-state* describes the next-state selector function, and *seln-R* describes the corresponding recursion relation.

definition

```

sel-next-state S == let (R,D)=S in case sel-next R D of None ⇒ None | Some t
⇒ Some (R,t#D)

```

definition

```

seln-R == graph sel-next-state

```

```

lemma seln-R-alt: seln-R == {((R,D),(R,t#D)) | R D t. sel-next R D = Some t}
  by (rule eq-reflection, unfold seln-R-def graph-def sel-next-state-def) (auto split:
option.split-asm)

```

8.3 Termination

8.3.1 Saturation upper bound

Before we can define the algorithm as recursive function, we have to prove termination, that is well-foundedness of the corresponding recursion relation *seln-R*

We start by defining a trivial finite upper bound for the saturation, simply as the set of all possible transitions in the automata. Intuitively, this bound is valid because the saturation algorithm only adds transitions, but never states to the automata

definition

```

seln-triv-upper R D == states D × ((fst o snd) ` (set R) ∪ alpha D) × states D

```

```

lemma seln-triv-upper-finite: finite (seln-triv-upper R D) by (unfold seln-triv-upper-def)
  (auto simp add: states-finite alpha-finite)

```

```

lemma D-below-triv-upper: set D ⊆ seln-triv-upper R D using statesAlpha-subset

```

```

  by (unfold seln-triv-upper-def) auto

```

```

lemma seln-triv-upper-subset-preserve: set D ⊆ seln-triv-upper A D' ==> seln-triv-upper
  A D ⊆ seln-triv-upper A D'

```

```

by (unfold seln-triv-upper-def) (blast intro: statesAlphaI dest: statesE alphaE)

lemma seln-triv-upper-mono: set D ⊆ set D'  $\implies$  seln-triv-upper R D ⊆ seln-triv-upper
R D'
by (unfold seln-triv-upper-def) (auto dest: states-mono alpha-mono)

lemma seln-triv-upper-mono-list: seln-triv-upper R D ⊆ seln-triv-upper R (t#D)
by (auto intro!: seln-triv-upper-mono)
lemma seln-triv-upper-mono-list': x ∈ seln-triv-upper R D  $\implies$  x ∈ seln-triv-upper R
(t#D) using seln-triv-upper-mono-list by (fast)

```

The trivial upper bound is not changed by inserting a transition to the automata that was already below the upper bound

```

lemma seln-triv-upper-inv: [t ∈ seln-triv-upper R D; set D' = insert t (set D)]  $\implies$ 
seln-triv-upper R D = seln-triv-upper R D'
by (unfold seln-triv-upper-def) (auto dest: statesAlpha-insert)

```

States returned by *find-sp* are valid states of the underlying automaton

```

lemma find-sp-in-states: find-sp D s p = Some qh  $\implies$  qh ∈ states D by (unfold
find-sp-def) (auto dest: first-thatE1 split: if-splits simp add: statesAlphaI)

```

The next-element selection function returns a new transition, that is below the trivial upper bound

```

lemma sel-next-below:
assumes A: sel-next R D = Some t
shows t ∉ set D  $\wedge$  t ∈ seln-triv-upper R D
proof –
{
  fix q a qh b q'
  assume A: (q, a, qh) ∈ set D and B: (qh, b, q') ∈ trcl (set D)
  from B statesAlpha-subset[of D] have q' ∈ states D
  apply –
  apply (erule (1) trcl-structE)
  using A by (simp-all add: statesAlphaI)
}
thus ?thesis
using A
apply (unfold sel-next-def seln-triv-upper-def)
apply (clar simp dest!: first-thatE1 lookupE1 split: if-splits option.split-asm)
apply (force simp add: find-sp-in-states dest!: first-thatE1 lookupE1 split: if-splits
option.split-asm)
done
qed

```

Hence, it does not change the upper bound

```

corollary sel-next-upper-preserve: [sel-next R D = Some t]  $\implies$  seln-triv-upper R
D = seln-triv-upper R (t#D) proof –
have set (t#D) = insert t (set D) by auto

```

```

moreover assume sel-next R D = Some t
with sel-next-below have t∈seln-triv-upper R D by blast
ultimately show ?thesis by (blast dest: seln-triv-upper-inv)
qed

```

8.3.2 Well-foundedness of recursion relation

```

lemma seln-R-wf: wf (seln-R-1) proof -
  let ?rel={((R,D),(R,D')) | R D D'. set D ⊂ set D' ∧ seln-triv-upper R D = seln-triv-upper R D'}
  have seln-R-1 ⊆ ?rel-1
    apply (unfold seln-R-alt)
    apply (clarify, safe)
    apply (blast dest: sel-next-below)
    apply (simp add: seln-triv-upper-mono-list')
    apply (simp add: sel-next-upper-preserve)
    done
  also
    let ?alpha=λx. let (R,D)=x in seln-triv-upper R D – set D
    let ?rel2=finite-psubset-1
    have ?rel-1 ⊆ inv-image (?rel2-1) ?alpha using D-below-triv-upper by (unfold finite-psubset-def, fastforce simp add: inv-image-def seln-triv-upper-finite)
    finally have seln-R-1 ⊆ inv-image (?rel2-1) ?alpha .
  moreover
    have wf (?rel2-1) using wf-finite-psubset by simp
    hence wf (inv-image (?rel2-1) ?alpha) by (rule wf-inv-image)
    ultimately show ?thesis by (blast intro: wf-subset)
qed

```

8.3.3 Definition of recursive function

```

function pss-algo-rec :: ('c,'s) seln-state ⇒ ('c,'s) seln-state
  where pss-algo-rec (R,D) = (case sel-next R D of Some t ⇒ pss-algo-rec (R,t#D)
  | None ⇒ (R,D))
    by pat-completeness auto

termination
  apply (relation seln-R-1)
  apply (simp add: seln-R-wf)
  unfolding seln-R-alt by blast

lemma pss-algo-rec-newsimps[simp]:
  [sel-next R D = None] ⇒ pss-algo-rec (R,D) = (R,D)
  [sel-next R D = Some t] ⇒ pss-algo-rec (R,D) = pss-algo-rec (R,t#D)
  by auto

declare pss-algo-rec.simps[simp del]

```

8.4 Correctness

8.4.1 *seln_R* refines *ps_R*

We show that *seln-R* refines *ps-R*, that is that every step made by our implementation corresponds to a step in the nondeterministic algorithm, that we already have proved correct in theory DPN.

```

lemma (in MFSM-ex) sel-nextE1:
  assumes A: sel-next R D = Some (s,γ,q')
  shows (s,γ,q')notin set D ∧ (exists q p a c'. s=sp A q p ∧ [p,γ]→a c' in rules M ∧
  (q,c',q')in trclAD A (set D))
  proof –
    let ?f = λp γ p' c' t. let (q,pp',sp') = t in
      if pp'=p' then
        case find-sp D q p of
          Some s ⇒ (case lookup (λq'. (s,γ,q')notin set D) D sp' c' of
            Some q' ⇒ Some (s,γ,q') |
            None ⇒ None
          ) | - ⇒ None
        else None
      else None

    let ?f1 = λr. let (p,γ,p',c') = r in first-that (?f p γ p' c') D

    from A[unfolded sel-next-def] obtain r where 1: r∈set R ∧ ?f1 r = Some
    (s,γ,q') by (blast dest: first-thatE1)
    then obtain p γh p' c' where 2: r=(p,γh,p',c') ∧ first-that (?f p γh p' c') D =
    Some (s,γ,q') by (cases r) simp
    then obtain t where 3: t∈set D ∧ ?f p γh p' c' t = Some (s,γ,q') by (blast
    dest: first-thatE1)
    then obtain q sp' where 4: t=(q,p',sp') ∧ (case find-sp D q p of
      Some s ⇒ (case lookup (λq'. (s,γh,q')notin set D) D sp' c' of
        Some q' ⇒ Some (s,γh,q') |
        None ⇒ None
      ) | - ⇒ None) = Some (s,γ,q')
    by (cases t, auto split: if-splits)
    hence 5: find-sp D q p = Some s ∧ lookup (λq'. (s,γh,q')notin set D) D sp' c' =
    Some q' ∧ γ=γh
    by (simp split: option.split-asm)
    with 1 2 rules-repr obtain a where 6: (p#[γ],a,p'#[c'])in rules M by (blast dest:
    rules-repr-cons)
    hence 7: p∈csyms M ∧ p'∈csyms M ∧ γ∈ssyms M by (blast dest: rule-fmt-fs)
    with 3 4 D-below have 8: q∈cstates A ∧ sp'=sp A q p' by (blast dest: csym-from-cstate'
    cstate-succ-unique')
    with 5 7 have 9: s=sp A q p using D-above D-below by (auto simp add:
    find-sp-cons)
    have 10: (s,γ,q')notin set D ∧ (sp',c',q')in trclAD A (set D) using 5 8 uniqueSp 7
    states-part D-below ps-upper-below-trivial
    apply – apply (rule lookup-trclAD-E1)
    by auto
  
```

moreover have $(q, p' \# c', q') \in \text{trclAD } A$ (set D) **proof** –
from 7 8 *sp-pred-ex* D -above **have** $(q, p', sp') \in \text{set } D$ **by auto**
with 10 *trclAD.cons* **show** ?*thesis* **using** 7 8 *alpha-cons states-part* **by auto**
qed
ultimately show ?*thesis* **using** 9 6 **by blast**
qed

lemma (in MFSM-ex) sel-nextE2:
assumes $A: \text{sel-next } R D = \text{None}$
shows $\neg(\exists q p \gamma q' a c' t. t \notin \text{set } D \wedge t = (sp A q p, \gamma, q') \wedge [p, \gamma] \hookrightarrow_a c' \in \text{rules } M \wedge (q, c', q') \in \text{trclAD } A$ (set D)
proof (clarify) — Assume we have such a rule and transition, and infer *sel-next R D* $\neq \text{None}$
fix $q p \gamma q' a pc'$
assume $C: (sp A q p, \gamma, q') \notin \text{set } D$ $([p, \gamma], a, pc') \in \text{rules } M$ $(q, pc', q') \in \text{trclAD } A$ (set D)
from C obtain $p' c'$ **where** *SYMS*: $p \in \text{csts M} \wedge p' \in \text{csts M} \wedge \gamma \in \text{ssyms M} \wedge pc' = p' \# c'$ **by** (blast dest: rule-fmt)
have *QCS*: $q \in \text{cstates } A$ $(q, p', sp A q p') \in \text{set } D$ $(sp A q p', c', q') \in \text{trclAD } A$ (set D) **proof** –
from C SYMS obtain sp' **where** $(q, p', sp') \in \text{set } D \wedge (sp', c', q') \in \text{trclAD } A$ (set D) **by** (blast dest: trclAD-uncons)
moreover with *D-below SYMS* **show** $q \in \text{cstates } A$ **by** (auto intro: csym-from-cstate')
ultimately show $(q, p', sp A q p') \in \text{set } D$ $(sp A q p', c', q') \in \text{trclAD } A$ (set D)
using *D-below cstate-succ-unique'* **by auto**
qed
from C QCS lookup-trclAD-II [of D set D $sp A q p' c' q' A (\lambda q''. (sp A q p, \gamma, q'') \notin \text{set } D)$] **obtain** q'' **where** *N1*: $\text{lookup } (\lambda q''. (sp A q p, \gamma, q'') \notin \text{set } D) D (sp A q p') c' = \text{Some } q''$ **by** blast
let $?f = \lambda p \gamma p' c' q pp' sp'$.
if $pp' = p'$ **then**
case *find-sp* $D q p$ **of**
 $\text{Some } s \Rightarrow (\text{case } \text{lookup } (\lambda q'. (s, \gamma, q') \notin \text{set } D) D sp' c' \text{ of}$
 $\text{Some } q' \Rightarrow \text{Some } (s, \gamma, q') \mid$
 $\text{None} \Rightarrow \text{None}$
 $) \mid \text{-} \Rightarrow \text{None}$
else None
from SYMS QCS have *FIND-SP*: $\text{find-sp } D q p = \text{Some } (sp A q p)$ **using**
D-below D-above **by** (simp add: *find-sp-cons*)
let $?f1 = (\lambda p \gamma p' c'. (\lambda t. \text{let } (q, pp', sp') = t \text{ in } ?f p \gamma p' c' q pp' sp'))$
from *N1 FIND-SP* **have** *N2*: $?f1 p \gamma p' c' (q, p', sp A q p') = \text{Some } (sp A q p, \gamma, q'')$ **by auto**
with *QCS first-thatII* [of $(q, p', sp A q p')$ D $?f1 p \gamma p' c'$] **obtain** t' **where** *N3*:

```

first-that (?f1 p γ p' c') D = Some t' by (blast)
let ?f2 = (λr. let (p,γ,p',c') = r in first-that (?f1 p γ p' c') D)
from N3 have ?f2 (p,γ,p',c') = Some t' by auto
moreover from SYMS C rules-repr have (p,γ,p',c')∈set R by (blast dest:
rules-repr-cons)
ultimately obtain t'' where first-that ?f2 R = Some t'' using first-thatI1[of
(p, γ, p', c') R ?f2] by (blast)
hence sel-next R D = Some t'' by (unfold sel-next-def)
with A show False by simp
qed

lemmas (in MFSM-ex) sel-nextE = sel-nextE1 sel-nextE2

lemma (in MFSM-ex) seln-cons1: [sel-next R D = Some t] ==> (set D,insert t (set
D))∈ps-R M A using D-below by (cases t, auto dest: sel-nextE intro: ps-R.intros)
lemma (in MFSM-ex) seln-cons2: sel-next R D = None ==> set D∉Domain (ps-R
M A) by (blast dest: sel-nextE elim: ps-R.cases)

lemma (in MFSM-ex) seln-cons1-rev: [set D∉Domain (ps-R M A)] ==> sel-next
R D = None by (cases sel-next R D) (auto iff add: seln-cons1 seln-cons2)
lemma (in MFSM-ex) seln-cons2-rev: [set D∈Domain (ps-R M A)] ==> ∃ t.
sel-next R D = Some t ∧ (set D,insert t (set D))∈ps-R M A
by (cases sel-next R D) (auto iff add: seln-cons1 seln-cons2 ps-R-dom-below)

DPN-specific abstraction relation, to associate states of deterministic algo-
rithm with states of ps-R

definition αseln M A == { (set D, (R,D)) | D R. MFSM-ex M A R D}

lemma αselnI: [S=set D; MFSM-ex M A R D] ==> (S,(R,D))∈αseln M A
by (unfold αseln-def) auto

lemma αselnD: (S,(R,D))∈αseln M A ==> S=set D ∧ MFSM-ex M A R D
by (unfold αseln-def) auto

lemma αselnD': (S,C)∈αseln M A ==> S=set (snd C) ∧ MFSM-ex M A (fst C)
(snd C) by (cases C, simp add: αselnD)

lemma αseln-single-valued: single-valued ((αseln M A)⁻¹)
by (unfold αseln-def) (auto intro: single-valuedI)

theorem (in MFSM) seln-refines: seln-R ≤αseln M A (ps-R M A) proof (rule
refinesI)
show αseln M A O seln-R ⊆ ps-R M A O αseln M A proof (rule refines-compI)
fix a c c'
assume ABS: (a,c)∈αseln M A and R: (c,c')∈seln-R
then obtain R D t where 1: c=(R,D) ∧ c'=(R,t#D) ∧ sel-next R D = Some
t by (unfold seln-R-alt, blast)
moreover with ABS have 2: a=set D ∧ MFSM-ex M A R D by (unfold
αseln-def, auto)

```

```

ultimately have 3: (set D,(set (t#D))) ∈ ps-R M A using MFSM-ex.seln-cons1[of
M A R D] by auto
  moreover have (set (t#D), (R,t#D)) ∈ αseln M A
  proof -
    from 2 have δ A ⊆ set D using MFSM-ex.D-above[of M A R D] by auto
    with 3 have δ A ⊆ set (t#D) set (t#D) ⊆ ps-upper M A using ps-R-below
  by (fast+)
    with 2 have MFSM-ex M A R (t#D) by (unfold MFSM-ex-alt, simp)
    thus ?thesis unfolding αseln-def by auto
  qed
  ultimately show ∃ a'. (a, a') ∈ ps-R M A ∧ (a', c') ∈ αseln M A using 1 2
  by blast
  qed
next
show αseln M A “ Domain (ps-R M A) ⊆ Domain seln-R
  apply (rule refines-domI)
  apply (unfold αseln-def seln-R-alt)
  apply (unfold Domain-iff)
  apply (clarify)
  apply (fast dest: MFSM-ex.seln-cons2-rev)
  done
qed

```

8.4.2 Computing transitions only

```

definition pss-algo :: 'c DPN-ex ⇒ ('s,'c) delta ⇒ ('s,'c) delta where
  pss-algo R D ≡ snd (pss-algo-rec (R,D))

```

8.4.3 Correctness

We have to show that the next-state selector function's graph refines *seln-R*. This is trivial because we defined *seln-R* to be that graph

```

lemma sns-refines: graph sel-next-state ≤Id seln-R by (unfold seln-R-def) simp

```

```

interpretation det-impl: detRef-impl pss-algo-rec sel-next-state seln-R
  apply (rule detRef-impl.intro)
  apply (simp-all only: detRef-wf-transfer[OF seln-R-wf] sns-refines)
  apply (unfold sel-next-state-def)
  apply (auto split: option.splits)
  done

```

And then infer correctness of the deterministic algorithm

```

theorem (in MFSM-ex) pss-correct:
  assumes D-init: set D = δ A
  shows lang (A( δ:=set (pss-algo R D) )) = pre-star (rules M) A
  proof (rule correct)
    have (set D, (R,D)) ∈ αseln M A by (intro refl αselnI) unfold-locale
    moreover have ((R,D),pss-algo-rec (R,D)) ∈ ndet-algo (seln-R) by (simp add:
      det-impl.algo-correct)
  
```

```

ultimately obtain d' where 1: (d',pss-algo-rec (R,D)) ∈ αseln M A ∧ (set
D,d') ∈ ndet-algo (ps-R M A) using refines-ndet-algo[OF seln-refines] by blast
hence d' = set (snd (pss-algo-rec (R,D))) by (blast dest: αselnD')
with 1 show (δ A, set (pss-algo R D)) ∈ ndet-algo (ps-R M A) using D-init
unfolding pss-algo-def by simp
qed

```

```

corollary (in MFSM) pss-correct:
assumes repr: set D = δ A (R,rules M) ∈ rules-repr
shows lang (A (δ := set (pss-algo R D))) = pre-star (rules M) A
proof –
  interpret MFSM-ex sep M M A R D
  apply simp-all
  apply unfold-locales
  apply (simp-all add: repr initial-delta-below)
  done
  from repr show ?thesis by (simp add: pss-correct)
qed

```

Generate executable code

```
export-code pss-algo checking SML
```

```
end
```

References

- [1] A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *Proc. of CONCUR'05*. Springer, 2005.