Stuttering Equivalence and Stuttering Invariance

Stephan Merz
Inria Nancy & LORIA
Villers-lès-Nancy, France

May 29, 2023

Two ω-sequences are stuttering equivalent if they differ only by finite repetitions of elements. For example, the two sequences

$$(abbccca)^\omega \qquad \text{and} \qquad (aaaabc)^\omega$$

are stuttering equivalent, whereas

$$(abac)^\omega \qquad \text{and} \qquad (aaaabcc)^\omega$$

are not. Stuttering equivalence is a fundamental concept in the theory of concurrent and distributed systems. Notably, Lamport [1] argues that refinement notions for such systems should be insensitive to finite stuttering. Peled and Wilke [2] showed that all PLTL (propositional linear-time temporal logic) properties that are insensitive to stuttering equivalence can be expressed without the next-time operator. Stuttering equivalence is also important for certain verification techniques such as partial-order reduction for model checking.

We formalize stuttering equivalence in Isabelle/HOL. Our development relies on the notion of stuttering sampling functions that may skip blocks of identical sequence elements. We also encode PLTL and prove the theorem due to Peled and Wilke [2].

Contents

theory *Samplers*
imports *Main HOL−Library.Omega-Words-Fun*
begin

1 Utility Lemmas

The following lemmas about strictly monotonic functions could go to the standard library of Isabelle/HOL.

Strongly monotonic functions over the integers grow without bound.

lemma *strict-mono-exceeds*:
assumes f: *strict-mono* (f::*nat* ⇒ *nat*)
shows ∃ k. $n < f\ k$
proof (*induct n*)
from f **have** $f\ 0 < f\ 1$ **by** (*rule strict-monoD*) *simp*
hence $0 < f\ 1$ **by** *simp*
thus ∃ k. $0 < f\ k$ **..**
next
fix n
assume ∃ k. $n < f\ k$
then obtain k **where** $n < f\ k$ **..**
hence *Suc* $n \leq f\ k$ **by** *simp*
also from f **have** $f\ k < f$ (*Suc* k) **by** (*rule strict-monoD*) *simp*
finally show ∃ k. *Suc* $n < f\ k$ **..**
qed

More precisely, any natural number $n \geq f\ 0$ lies in the interval between $f\ k$ and f (*Suc* k), for some k.

lemma *strict-mono-interval*:
assumes f: *strict-mono* (f::*nat* ⇒ *nat*) **and** n: $f\ 0 \leq n$
obtains k **where** $f\ k \leq n$ **and** $n < f$ (*Suc* k)
proof −
from f[*THEN strict-mono-exceeds*] **obtain** m **where** m: $n < f\ m$ **..**
have $m \neq 0$
proof
assume $m = 0$
with m n **show** *False* **by** *simp*
qed
with m **obtain** m' **where** m': $n < f$ (*Suc* m') **by** (*auto simp*: *gr0-conv-Suc*)
let *?k* = *LEAST* k. $n < f$ (*Suc* k)

```
  from m' have 1: n < f (Suc ?k) by (rule LeastI)
  have f ?k ≤ n
  proof (rule ccontr)
    assume ¬ ?thesis
    hence k: n < f ?k by simp
    show False
    proof (cases ?k)
      case 0 with k n show False by simp
    next
      case Suc with k show False by (auto dest: Least-le)
    qed
  qed
  with 1 that show ?thesis by simp
qed

lemma strict-mono-comp:
  assumes g: strict-mono (g::'a::order ⇒ 'b::order)
      and f: strict-mono (f::'b::order ⇒ 'c::order)
  shows strict-mono (f ∘ g)
  using assms by (auto simp: strict-mono-def)
```

2 Stuttering Sampling Functions

Given an ω-sequence σ, a stuttering sampling function is a strictly monotonic function $f::nat \Rightarrow nat$ such that $f\ 0 = 0$ and for all i and all $f\ i \leq k < f\ (i+1)$, the elements $\sigma\ k$ are the same. In other words, f skips some (but not necessarily all) stuttering steps, but never skips a non-stuttering step. Given such σ and f, the (stuttering-)sampled reduction of σ is the sequence of elements of σ at the indices $f\ i$, which can simply be written as $\sigma \circ f$.

2.1 Definition and elementary properties

```
definition stutter-sampler where
  — f is a stuttering sampling function for σ
  stutter-sampler (f::nat ⇒ nat) σ ≡
     f 0 = 0
   ∧ strict-mono f
   ∧ (∀ k n. f k < n ∧ n < f (Suc k) ⟶ σ n = σ (f k))

lemma stutter-sampler-0: stutter-sampler f σ ⟹ f 0 = 0
  by (simp add: stutter-sampler-def)

lemma stutter-sampler-mono: stutter-sampler f σ ⟹ strict-mono f
  by (simp add: stutter-sampler-def)

lemma stutter-sampler-between:
  assumes f: stutter-sampler f σ
      and lo: f k ≤ n and hi: n < f (Suc k)
```

shows $\sigma\ n = \sigma\ (f\ k)$
using *assms* **by** (*auto simp*: *stutter-sampler-def less-le*)

lemma *stutter-sampler-interval*:
assumes *f*: *stutter-sampler f* σ
obtains *k* **where** $f\ k \leq n$ **and** $n < f\ (Suc\ k)$
using *f*[*THEN stutter-sampler-mono*] **proof** (*rule strict-mono-interval*)
from *f* **show** $f\ 0 \leq n$ **by** (*simp add*: *stutter-sampler-0*)
qed

The identity function is a stuttering sampling function for any σ.

lemma *id-stutter-sampler* [*iff*]: *stutter-sampler id* σ
by (*auto simp*: *stutter-sampler-def strict-mono-def*)

Stuttering sampling functions compose, sort of.

lemma *stutter-sampler-comp*:
assumes *f*: *stutter-sampler f* σ
and *g*: *stutter-sampler g* $(\sigma \circ f)$
shows *stutter-sampler* $(f \circ g)\ \sigma$
proof (*auto simp*: *stutter-sampler-def*)
from *f g* **show** $f\ (g\ 0) = 0$ **by** (*simp add*: *stutter-sampler-0*)
next
from *g*[*THEN stutter-sampler-mono*] *f*[*THEN stutter-sampler-mono*]
show *strict-mono* $(f \circ g)$ **by** (*rule strict-mono-comp*)
next
fix *i k*
assume *lo*: $f\ (g\ i) < k$ **and** *hi*: $k < f\ (g\ (Suc\ i))$
from *f* **obtain** *m* **where** *1*: $f\ m \leq k$ **and** *2*: $k < f\ (Suc\ m)$
by (*rule stutter-sampler-interval*)
with *f* **have** *3*: $\sigma\ k = \sigma\ (f\ m)$ **by** (*rule stutter-sampler-between*)
from *lo 2* **have** $f\ (g\ i) < f\ (Suc\ m)$ **by** *simp*
with *f*[*THEN stutter-sampler-mono*] **have** *4*: $g\ i \leq m$ **by** (*simp add*: *strict-mono-less*)
from *1 hi* **have** $f\ m < f\ (g\ (Suc\ i))$ **by** *simp*
with *f*[*THEN stutter-sampler-mono*] **have** *5*: $m < g\ (Suc\ i)$**by** (*simp add*: *strict-mono-less*)
from *g 4 5* **have** $(\sigma \circ f)\ m = (\sigma \circ f)\ (g\ i)$ **by** (*rule stutter-sampler-between*)
with *3* **show** $\sigma\ k = \sigma\ (f\ (g\ i))$ **by** *simp*
qed

Stuttering sampling functions can be extended to suffixes.

lemma *stutter-sampler-suffix*:
assumes *f*: *stutter-sampler f* σ
shows *stutter-sampler* $(\lambda k.\ f\ (n+k) - f\ n)$ (*suffix* $(f\ n)\ \sigma$)
proof (*auto simp*: *stutter-sampler-def strict-mono-def*)
fix *i j*
assume *ij*: $(i::nat) < j$
from *f* **have** *mono*: *strict-mono f* **by** (*rule stutter-sampler-mono*)

from *mono*[*THEN strict-mono-mono*] **have** $f\ n \leq f\ (n+i)$

```
    by (rule monoD) simp
  moreover
  from mono[THEN strict-mono-mono] have f n ≤ f (n+j)
    by (rule monoD) simp
  moreover
  from mono ij have f (n+i) < f (n+j) by (auto intro: strict-monoD)
  ultimately
  show f (n+i) − f n < f (n+j) − f n by simp
next
  fix i k
  assume lo: f (n+i) − f n < k and hi: k < f (Suc (n+i)) − f n
  from lo have f (n+i) ≤ f n + k by simp
  moreover
  from hi have f n + k < f (Suc (n + i)) by simp
  moreover
  from f[THEN stutter-sampler-mono, THEN strict-mono-mono]
  have f n ≤ f (n+i) by (rule monoD) simp
  ultimately show σ (f n + k) = σ (f n + (f (n+i) − f n))
    by (auto dest: stutter-sampler-between[OF f])
qed
```

2.2 Preservation of properties through stuttering sampling

Stuttering sampling preserves the initial element of the sequence, as well as the presence and relative ordering of different elements.

```
lemma stutter-sampled-0:
  assumes stutter-sampler f σ
  shows σ (f 0) = σ 0
  using assms[THEN stutter-sampler-0] by simp

lemma stutter-sampled-in-range:
  assumes f: stutter-sampler f σ and s: s ∈ range σ
  shows s ∈ range (σ ∘ f)
proof −
  from s obtain n where n: σ n = s by auto
  from f obtain k where f k ≤ n n < f (Suc k) by (rule stutter-sampler-interval)
  with f have σ n = σ (f k) by (rule stutter-sampler-between)
  with n show ?thesis by auto
qed

lemma stutter-sampled-range:
  range (σ ∘ f) = range σ if stutter-sampler f σ
  using that stutter-sampled-in-range [of f σ] by auto

lemma stutter-sampled-precedence:
  assumes f: stutter-sampler f σ and ij: i ≤ j
  obtains k l where k ≤ l σ (f k) = σ i σ (f l) = σ j
proof −
  from f obtain k where k: f k ≤ i i < f (Suc k) by (rule stutter-sampler-interval)
```

```
  with f have 1: σ i = σ (f k) by (rule stutter-sampler-between)
  from f obtain l where l: f l ≤ j j < f (Suc l) by (rule stutter-sampler-interval)
  with f have 2: σ j = σ (f l) by (rule stutter-sampler-between)
  from k l ij have f k < f (Suc l) by simp
  with f[THEN stutter-sampler-mono] have k ≤ l by (simp add: strict-mono-less)
  with 1 2 that show ?thesis by simp
qed
```

2.3 Maximal stuttering sampling

We define a particular sampling function that is maximal in the sense that it eliminates all finite stuttering. If a sequence ends with infinite stuttering then it behaves as the identity over the (maximal such) suffix.

```
fun max-stutter-sampler where
  max-stutter-sampler σ 0 = 0
| max-stutter-sampler σ (Suc n) =
    (let prev = max-stutter-sampler σ n
     in  if (∀ k > prev. σ k = σ prev)
         then Suc prev
         else (LEAST k. prev < k ∧ σ k ≠ σ prev))
```

max-stutter-sampler is indeed a stuttering sampling function.

```
lemma max-stutter-sampler:
  stutter-sampler (max-stutter-sampler σ) σ (is stutter-sampler ?ms -)
proof −
  have ?ms 0 = 0 by simp
  moreover
  have ∀ n. ?ms n < ?ms (Suc n)
  proof
    fix n
    show ?ms n < ?ms (Suc n) (is ?prev < ?next)
    proof (cases ∀ k > ?prev. σ k = σ ?prev)
      case True thus ?thesis by (simp add: Let-def)
    next
      case False
      hence ∃ k. ?prev < k ∧ σ k ≠ σ ?prev by simp
      from this[THEN LeastI-ex]
      have ?prev < (LEAST k. ?prev < k ∧ σ k ≠ σ ?prev) ..
      with False show ?thesis by (simp add: Let-def)
    qed
  qed
  hence strict-mono ?ms
    unfolding strict-mono-def by (blast intro: lift-Suc-mono-less)
  moreover
  have ∀ n k. ?ms n < k ∧ k < ?ms (Suc n) ⟶ σ k = σ (?ms n)
  proof (clarify)
    fix n k
    assume lo: ?ms n < k (is ?prev < k)
```

```
    and hi: k < ?ms (Suc n) (is k < ?next)
  show σ k = σ ?prev
  proof (cases ∀ k > ?prev. σ k = σ ?prev)
    case True
    hence ?next = Suc ?prev by (simp add: Let-def)
    with lo hi show ?thesis by simp  — no room for intermediate index
  next
    case False
    hence ?next = (LEAST k. ?prev < k ∧ σ k ≠ σ ?prev)
      by (auto simp add: Let-def)
    with lo hi show ?thesis by (auto dest: not-less-Least)
  qed
 qed
 ultimately show ?thesis unfolding stutter-sampler-def by blast
qed
```

We write $\natural\sigma$ for the sequence σ sampled by the maximal stuttering sampler. Also, a sequence is *stutter free* if it contains no finite stuttering: whenever two subsequent elements are equal then all subsequent elements are the same.

```
definition stutter-reduced (♮- [100] 100) where
 ♮σ = σ ∘ (max-stutter-sampler σ)

definition stutter-free where
 stutter-free σ ≡ ∀ k. σ (Suc k) = σ k ⟶ (∀ n>k. σ n = σ k)

lemma stutter-freeI:
 assumes ⋀k n. ⟦σ (Suc k) = σ k; n>k⟧ ⟹ σ n = σ k
 shows stutter-free σ
 using assms unfolding stutter-free-def by blast

lemma stutter-freeD:
 assumes stutter-free σ and σ (Suc k) = σ k and n>k
 shows σ n = σ k
 using assms unfolding stutter-free-def by blast
```

Any suffix of a stutter free sequence is itself stutter free.

```
lemma stutter-free-suffix:
 assumes sigma: stutter-free σ
 shows stutter-free (suffix k σ)
proof (rule stutter-freeI)
 fix j n
 assume j: (suffix k σ) (Suc j) = (suffix k σ) j and n: j < n
 from j have σ (Suc (k+j)) = σ (k+j) by simp
 moreover from n have k+n > k+j by simp
 ultimately have σ (k+n) = σ (k+j) by (rule stutter-freeD[OF sigma])
 thus (suffix k σ) n = (suffix k σ) j by simp
qed
```

```
lemma stutter-reduced-0: (♮σ) 0 = σ 0
  by (simp add: stutter-reduced-def stutter-sampled-0 max-stutter-sampler)

lemma stutter-free-reduced:
  assumes sigma: stutter-free σ
  shows ♮σ = σ
proof −
  {
    fix n
    have max-stutter-sampler σ n = n (is ?ms n = n)
    proof (induct n)
      show ?ms 0 = 0 by simp
    next
      fix n
      assume ih: ?ms n = n
      show ?ms (Suc n) = Suc n
      proof (cases σ (Suc n) = σ (?ms n))
        case True
        with ih have σ (Suc n) = σ n by simp
        with sigma have ∀ k > n. σ k = σ n
          unfolding stutter-free-def by blast
        with ih show ?thesis by (simp add: Let-def)
      next
        case False
        with ih have (LEAST k. k>n ∧ σ k ≠ σ (?ms n)) = Suc n
          by (auto intro: Least-equality)
        with ih False show ?thesis by (simp add: Let-def)
      qed
    qed
  }
  thus ?thesis by (auto simp: stutter-reduced-def)
qed
```

Whenever two sequence elements at two consecutive sampling points of the maximal stuttering sampler are equal then the sequence stutters infinitely from the first sampling point onwards. In particular, ♮σ is stutter free.

```
lemma max-stutter-sampler-nostuttering:
  assumes stut: σ (max-stutter-sampler σ (Suc k)) = σ (max-stutter-sampler σ k)
      and n: n > max-stutter-sampler σ k (is - > ?ms k)
  shows σ n = σ (?ms k)
proof (rule ccontr)
  assume contr: ¬ ?thesis
  with n have ?ms k < n ∧ σ n ≠ σ (?ms k) (is ?diff n) ..
  hence ?diff (LEAST n. ?diff n) by (rule LeastI)
  with contr have σ (?ms (Suc k)) ≠ σ (?ms k) by (auto simp add: Let-def)
  from this stut show False ..
qed

lemma stutter-reduced-stutter-free: stutter-free (♮σ)
```

```
proof (rule stutter-freeI)
  fix k n
  assume k: (♮σ) (Suc k) = (♮σ) k and n: k < n
  from n have max-stutter-sampler σ k < max-stutter-sampler σ n
    using max-stutter-sampler[THEN stutter-sampler-mono, THEN strict-monoD]
    by blast
  with k show (♮σ) n = (♮σ) k
    unfolding stutter-reduced-def
    by (auto elim: max-stutter-sampler-nostuttering
          simp del: max-stutter-sampler.simps)
qed

lemma stutter-reduced-suffix: ♮ (suffix k (♮σ)) = suffix k (♮σ)
proof (rule stutter-free-reduced)
  have stutter-free (♮σ) by (rule stutter-reduced-stutter-free)
  thus stutter-free (suffix k (♮σ)) by (rule stutter-free-suffix)
qed

lemma stutter-reduced-reduced: ♮♮σ = ♮σ
  by (insert stutter-reduced-suffix[of 0 σ, simplified])
```

One can define a partial order on sampling functions for a given sequence σ by saying that function g is better than function f if the reduced sequence induced by f can be further reduced to obtain the reduced sequence corresponding to g, i.e. if there exists a stuttering sampling function h for the reduced sequence $\sigma \circ f$ such that $\sigma \circ f \circ h = \sigma \circ g$. (Note that $f \circ h$ is indeed a stuttering sampling function for σ, by theorem *stutter-sampler-comp*.)

We do not formalize this notion but prove that *max-stutter-sampler* σ is the best sampling function according to this order.

```
theorem sample-max-sample:
  assumes f: stutter-sampler f σ
  shows ♮(σ ∘ f) = ♮σ
proof −
  let ?mss = max-stutter-sampler σ
  let ?mssf = max-stutter-sampler (σ ∘ f)
  from f have mssf: stutter-sampler (f ∘ ?mssf) σ
    by (blast intro: stutter-sampler-comp max-stutter-sampler)
```

The following is the core invariant of the proof: the sampling functions *max-stutter-sampler* σ and $f \circ$ (*max-stutter-sampler* $(\sigma \circ f)$) work in lock-step (i.e., sample the same points), except if σ ends in infinite stuttering, at which point function f may make larger steps than the maximal sampling functions.

```
  {
    fix k
    have   ?mss k = f (?mssf k)
         ∨ ?mss k ≤ f (?mssf k) ∧ (∀ n ≥ ?mss k. σ (?mss k) = σ n)
         (is ?P k is ?A k ∨ ?B k)
    proof (induct k)
```

```
  from f mssf have ?mss 0 = f (?mssf 0)
    by (simp add: max-stutter-sampler stutter-sampler-0)
  thus ?P 0 ..
next
  fix k
  assume ih: ?P k
  have b: ?B k ⟶ ?B (Suc k)
  proof
    assume 0: ?B k hence 1: ?mss k ≤ f (?mssf k) ..

    from 0 have 2: ∀ n ≥ ?mss k. σ (?mss k) = σ n ..
    hence ∀ n > ?mss k. σ (?mss k) = σ n by auto
    hence ∀ n > ?mss k. σ n = σ (?mss k) by auto
    hence 3: ?mss (Suc k) = Suc (?mss k) by (simp add: Let-def)
    with 2 have σ (?mss k) = σ (?mss (Suc k))
      by (auto simp del: max-stutter-sampler.simps)
    from sym[OF this] 2 3 have ∀ n ≥ ?mss (Suc k). σ (?mss (Suc k)) = σ n
      by (auto simp del: max-stutter-sampler.simps)
    moreover
    from mssf[THEN stutter-sampler-mono, THEN strict-monoD]
    have f (?mssf k) < f (?mssf (Suc k))
      by (simp del: max-stutter-sampler.simps)
    with 1 3 have ?mss (Suc k) ≤ f (?mssf (Suc k))
      by (simp del: max-stutter-sampler.simps)
    ultimately show ?B (Suc k) by blast
  qed
  from ih show ?P (Suc k)
  proof
    assume a: ?A k
    show ?thesis
    proof (cases ∀ n > ?mss k. σ n = σ (?mss k))
      case True
      hence ∀ n ≥ ?mss k. σ (?mss k) = σ n by (auto simp: le-less)
      with a have ?B k by simp
      with b show ?thesis by (simp del: max-stutter-sampler.simps)
    next
      case False
      hence diff: σ (?mss (Suc k)) ≠ σ (?mss k)
        by (blast dest: max-stutter-sampler-nostuttering)
      have ?A (Suc k)
      proof (rule antisym)
        show f (?mssf (Suc k)) ≤ ?mss (Suc k)
        proof (rule ccontr)
          assume ¬ ?thesis
          hence contr: ?mss (Suc k) < f (?mssf (Suc k)) by simp
          from mssf have σ (?mss (Suc k)) = σ ((f ∘ ?mssf) k)
          proof (rule stutter-sampler-between)
            from max-stutter-sampler[of σ, THEN stutter-sampler-mono]
            have ?mss k < ?mss (Suc k) by (rule strict-monoD) simp
```

```
    with a show (f ∘ ?mssf) k ≤ ?mss (Suc k)
      by (simp add: o-def del: max-stutter-sampler.simps)
  next
    from contr show ?mss (Suc k) < (f ∘ ?mssf) (Suc k) by simp
  qed
  with a have σ (?mss (Suc k)) = σ (?mss k)
    by (simp add: o-def del: max-stutter-sampler.simps)
  with diff show False ..
 qed
next
 have ∃ m > ?mssf k. f m = ?mss (Suc k)
 proof (rule ccontr)
  assume ¬ ?thesis
  hence contr: ∀ i. f ((?mssf k) + Suc i) ≠ ?mss (Suc k) by simp
  {
   fix i
   have f (?mssf k + i) < ?mss (Suc k) (is ?F i)
   proof (induct i)
    from a have f (?mssf k + 0) = ?mss k by (simp add: o-def)
   also from max-stutter-sampler[of σ, THEN stutter-sampler-mono]
        have ... < ?mss (Suc k)
          by (rule strict-monoD) simp
    finally show ?F 0 .
   next
    fix i
    assume ih: ?F i
    show ?F (Suc i)
    proof (rule ccontr)
     assume ¬ ?thesis
     then have ?mss (Suc k) ≤ f (?mssf k + Suc i)
       by (simp add: o-def)
     moreover from contr have f (?mssf k + Suc i) ≠ ?mss (Suc k)
       by blast
     ultimately have i: ?mss (Suc k) < f (?mssf k + Suc i)
       by (simp add: less-le)
     from f have σ (?mss (Suc k)) = σ (f (?mssf k + i))
     proof (rule stutter-sampler-between)
      from ih show f (?mssf k + i) ≤ ?mss (Suc k)
        by (simp add: o-def)
     next
      from i show ?mss (Suc k) < f (Suc (?mssf k + i))
        by simp
     qed
     also from max-stutter-sampler have ... = σ (?mss k)
     proof (rule stutter-sampler-between)
      from f[THEN stutter-sampler-mono, THEN strict-mono-mono]
      have f (?mssf k) ≤ f (?mssf k + i) by (rule monoD) simp
      with a show ?mss k ≤ f (?mssf k + i) by (simp add: o-def)
     qed (rule ih)
```

```
            also note diff
            finally show False by simp
          qed
        qed
      } note bounded = this
      from f[THEN stutter-sampler-mono]
      have strict-mono (λi. f (?mssf k + i))
        by (auto simp: strict-mono-def)
      then obtain i where i: ?mss (Suc k) < f (?mssf k + i)
        by (blast dest: strict-mono-exceeds)
      from bounded have f (?mssf k + i) < ?mss (Suc k) .
      with i show False by (simp del: max-stutter-sampler.simps)
    qed
    then obtain m where m: m > ?mssf k and m': f m = ?mss (Suc k)
      by blast
    show ?mss (Suc k) ≤ f (?mssf (Suc k))
    proof (rule ccontr)
      assume ¬ ?thesis
      hence contr: f (?mssf (Suc k)) < ?mss (Suc k) by simp
      from mssf[THEN stutter-sampler-mono]
      have (f ∘ ?mssf) k < (f ∘ ?mssf) (Suc k)
        by (rule strict-monoD) simp
      with a have ?mss k ≤ f (?mssf (Suc k))
        by (simp add: o-def)
      from this contr have σ (f (?mssf (Suc k))) = σ (?mss k)
        by (rule stutter-sampler-between[OF max-stutter-sampler])
      with a have stut: (σ ∘ f) (?mssf (Suc k)) = (σ ∘ f) (?mssf k)
        by (simp add: o-def)
      from this m have (σ ∘ f) m = (σ ∘ f) (?mssf k)
        by (blast intro: max-stutter-sampler-nostuttering)
      with diff m' a show False
        by (simp add: o-def)
    qed
  qed
  thus ?thesis ..
 qed
next
 assume ?B k with b show ?thesis by (simp del: max-stutter-sampler.simps)
qed
qed
}
hence ♮σ = ♮(σ ∘ f) unfolding stutter-reduced-def by force
thus ?thesis by (rule sym)
qed


end
theory StutterEquivalence
imports Samplers
```

begin

3 Stuttering Equivalence

Stuttering equivalence of two sequences is formally defined as the equality of their maximally reduced versions.

definition *stutter-equiv* (**infix** $\approx$ *50*) **where**
  $\sigma \approx \tau \equiv \natural\sigma = \natural\tau$

Stuttering equivalence is an equivalence relation.

lemma *stutter-equiv-refl*: $\sigma \approx \sigma$
  unfolding *stutter-equiv-def* **..**

lemma *stutter-equiv-sym* [*sym*]: $\sigma \approx \tau \Longrightarrow \tau \approx \sigma$
  unfolding *stutter-equiv-def* **by** (*rule sym*)

lemma *stutter-equiv-trans* [*trans*]: $\varrho \approx \sigma \Longrightarrow \sigma \approx \tau \Longrightarrow \varrho \approx \tau$
  unfolding *stutter-equiv-def* **by** *simp*

In particular, any sequence sampled by a stuttering sampler is stuttering equivalent to the original one.

lemma *sampled-stutter-equiv*:
  assumes *stutter-sampler f* σ
  shows $\sigma \circ f \approx \sigma$
  using *assms* **unfolding** *stutter-equiv-def* **by** (*rule sample-max-sample*)

lemma *stutter-reduced-equivalent*: $\natural\sigma \approx \sigma$
  unfolding *stutter-equiv-def* **by** (*rule stutter-reduced-reduced*)

For proving stuttering equivalence of two sequences, it is enough to exhibit two arbitrary sampling functions that equalize the reductions of the sequences. This can be more convenient than computing the maximal stutter-reduced version of the sequences.

lemma *stutter-equivI*:
  assumes *f*: *stutter-sampler f* σ **and** *g*: *stutter-sampler g* τ
    and *eq*: $\sigma \circ f = \tau \circ g$
  shows $\sigma \approx \tau$
proof −
  from *f* **have** $\natural\sigma = \natural(\sigma \circ f)$ **by** (*rule sample-max-sample*[*THEN sym*])
  also from *eq* **have** ... $= \natural(\tau \circ g)$ **by** *simp*
  also from *g* **have** ... $= \natural\tau$ **by** (*rule sample-max-sample*)
  finally show *?thesis* **by** (*unfold stutter-equiv-def*)
qed

The corresponding elimination rule is easy to prove, given that the maximal stuttering sampling function is a stuttering sampling function.

lemma *stutter-equivE*:
assumes *eq*: $\sigma \approx \tau$
and *p*: $\bigwedge f\ g.\ [\![\ \textit{stutter-sampler}\ f\ \sigma;\ \textit{stutter-sampler}\ g\ \tau;\ \sigma \circ f = \tau \circ g\ ]\!] \Longrightarrow P$
shows P
proof (*rule p*)
from *eq* **show** $\sigma \circ (\textit{max-stutter-sampler}\ \sigma) = \tau \circ (\textit{max-stutter-sampler}\ \tau)$
by (*unfold stutter-equiv-def stutter-reduced-def*)
qed (*rule max-stutter-sampler*)+

Therefore we get the following alternative characterization: two sequences are stuttering equivalent iff there are stuttering sampling functions that equalize the two sequences.

lemma *stutter-equiv-eq*:
$\sigma \approx \tau = (\exists f\ g.\ \textit{stutter-sampler}\ f\ \sigma \wedge \textit{stutter-sampler}\ g\ \tau \wedge \sigma \circ f = \tau \circ g)$
by (*blast intro*: *stutter-equivI elim*: *stutter-equivE*)

The initial elements of stutter equivalent sequences are equal.

lemma *stutter-equiv-0*:
assumes $\sigma \approx \tau$
shows $\sigma\ 0 = \tau\ 0$
proof −
have $\sigma\ 0 = (\natural\sigma)\ 0$ **by** (*rule stutter-reduced-0*[*THEN sym*])
with *assms*[*unfolded stutter-equiv-def*] **show** *?thesis*
by (*simp add*: *stutter-reduced-0*)
qed

abbreviation *suffix-notation* (- [-..])
where
suffix-notation w k ≡ *suffix k w*

Given any stuttering sampling function f for sequence σ, any suffix of σ starting at index $f\ n$ is stuttering equivalent to the suffix of the stutter-reduced version of σ starting at n.

lemma *suffix-stutter-equiv*:
assumes *f*: *stutter-sampler f* σ
shows *suffix* (*f n*) $\sigma \approx$ *suffix n* ($\sigma \circ f$)
proof −
from *f* **have** *stutter-sampler* ($\lambda k.\ f\ (n+k) - f\ n$) ($\sigma$[*f n* ..])
by (*rule stutter-sampler-suffix*)
moreover
have *stutter-sampler id* (($\sigma \circ f$)[*n* ..])
by (*rule id-stutter-sampler*)
moreover
have (σ[*f n* ..]) ∘ ($\lambda k.\ f\ (n+k) - f\ n$) = (($\sigma \circ f$)[*n* ..]) ∘ *id*
proof (*rule ext*, *auto*)
fix *i*
from *f*[*THEN stutter-sampler-mono*, *THEN strict-mono-mono*]
have *f n* ≤ *f* (*n*+*i*) **by** (*rule monoD*) *simp*

```
    thus σ (f n + (f (n+i) − f n)) = σ (f (n+i)) by simp
  qed
  ultimately show ?thesis
    by (rule stutter-equivI)
qed
```

Given a stuttering sampling function f and a point n within the interval from $f\ k$ to $f\ (k+1)$, the suffix starting at n is stuttering equivalent to the suffix starting at $f\ k$.

```
lemma stutter-equiv-within-interval:
  assumes f: stutter-sampler f σ
      and lo: f k ≤ n and hi: n < f (Suc k)
  shows σ[n ..] ≈ σ[f k ..]
proof −
  have stutter-sampler id (σ[n ..]) by (rule id-stutter-sampler)
  moreover
  from lo have stutter-sampler (λi. if i=0 then 0 else n + i − f k) (σ[f k ..])
    (is stutter-sampler ?f -)
  proof (auto simp: stutter-sampler-def strict-mono-def)
    fix i
    assume i: i < Suc n − f k
    from f show σ (f k + i) = σ (f k)
    proof (rule stutter-sampler-between)
      from i hi show f k + i < f (Suc k) by simp
    qed simp
  qed
  moreover
  have (σ[n ..]) ∘ id = (σ[f k ..]) ∘ ?f
  proof (rule ext, auto)
    from f lo hi show σ n = σ (f k) by (rule stutter-sampler-between)
  next
    fix i
    from lo show σ (n+i) = σ (f k + (n + i − f k)) by simp
  qed
  ultimately show ?thesis by (rule stutter-equivI)
qed
```

Given two stuttering equivalent sequences σ and τ, we obtain a zig-zag relationship as follows: for any suffix $\tau[n..]$ there is a suffix $\sigma[m..]$ such that:

1. $\sigma[m..] \approx \tau[n..]$ and
2. for every suffix $\sigma[j..]$ where $j<m$ there is a corresponding suffix $\tau[k..]$ for some $k<n$.

```
theorem stutter-equiv-suffixes-left:
  assumes σ ≈ τ
  obtains m where σ[m..] ≈ τ[n..] and ∀ j<m. ∃ k<n. σ[j..] ≈ τ[k..]
using assms proof (rule stutter-equivE)
```

```
  fix f g
  assume f: stutter-sampler f σ
     and g: stutter-sampler g τ
     and eq: σ ∘ f = τ ∘ g
  from g obtain i where i: g i ≤ n n < g (Suc i)
    by (rule stutter-sampler-interval)
  with g have τ[n..] ≈ τ[g i ..]
    by (rule stutter-equiv-within-interval)
  also from g have ... ≈ (τ ∘ g)[i ..]
    by (rule suffix-stutter-equiv)
  also from eq have ... = (σ ∘ f)[i ..]
    by simp
  also from f have ... ≈ σ[f i ..]
    by (rule suffix-stutter-equiv[THEN stutter-equiv-sym])
  finally have σ[f i ..] ≈ τ[n ..]
    by (rule stutter-equiv-sym)
  moreover
  {
    fix j
    assume j: j < f i
    from f obtain a where a: f a ≤ j j < f (Suc a)
      by (rule stutter-sampler-interval)
    from a j have f a < f i by simp
    with f[THEN stutter-sampler-mono] have a < i
      by (simp add: strict-mono-less)
    with g[THEN stutter-sampler-mono] have g a < g i
      by (simp add: strict-mono-less)
    with i have 1: g a < n by simp

    from f a have σ[j..] ≈ σ[f a ..]
      by (rule stutter-equiv-within-interval)
    also from f have ... ≈ (σ ∘ f)[a ..]
      by (rule suffix-stutter-equiv)
    also from eq have ... = (τ ∘ g)[a ..] by simp
    also from g have ... ≈ τ[g a ..]
      by (rule suffix-stutter-equiv[THEN stutter-equiv-sym])
    finally have σ[j ..] ≈ τ[g a ..] .
    with 1 have ∃k<n. σ[j..] ≈ τ[k ..] by blast
  }
  moreover
  note that
  ultimately show ?thesis by blast
qed

theorem stutter-equiv-suffixes-right:
  assumes σ ≈ τ
  obtains n where σ[m..] ≈ τ[n..] and ∀j<n. ∃k<m. σ[k..] ≈ τ[j..]
proof −
  from assms have τ ≈ σ
```

by (*rule stutter-equiv-sym*)
then obtain n **where** $\tau[n..] \approx \sigma[m..]$ $\forall j{<}n.\ \exists k{<}m.\ \tau[j..] \approx \sigma[k..]$
by (*rule stutter-equiv-suffixes-left*)
with *that* **show** *?thesis*
by (*blast dest*: *stutter-equiv-sym*)
qed

In particular, if σ and τ are stutter equivalent then every element that occurs in one sequence also occurs in the other.

lemma *stutter-equiv-element-left*:
assumes $\sigma \approx \tau$
obtains m **where** $\sigma\ m = \tau\ n$ **and** $\forall j{<}m.\ \exists k{<}n.\ \sigma\ j = \tau\ k$
proof −
from *assms* **obtain** m **where** $\sigma[m..] \approx \tau[n..]$ $\forall j{<}m.\ \exists k{<}n.\ \sigma[j..] \approx \tau[k..]$
by (*rule stutter-equiv-suffixes-left*)
with *that* **show** *?thesis*
by (*force dest*: *stutter-equiv-0*)
qed

lemma *stutter-equiv-element-right*:
assumes $\sigma \approx \tau$
obtains n **where** $\sigma\ m = \tau\ n$ **and** $\forall j{<}n.\ \exists k{<}m.\ \sigma\ k = \tau\ j$
proof −
from *assms* **obtain** n **where** $\sigma[m..] \approx \tau[n..]$ $\forall j{<}n.\ \exists k{<}m.\ \sigma[k..] \approx \tau[j..]$
by (*rule stutter-equiv-suffixes-right*)
with *that* **show** *?thesis*
by (*force dest*: *stutter-equiv-0*)
qed

end
theory *PLTL*
imports *Main LTL.LTL Samplers StutterEquivalence*
begin

4 Stuttering Invariant LTL Formulas

We show that the next-free fragment of propositional linear-time temporal logic PLTL is invariant to finite stuttering.

4.1 Finite Conjunctions and Disjunctions in PLTL

definition *OR* **where** *OR* $\Phi \equiv$ *SOME* φ. *fold-graph Or-ltlp False-ltlp* Φ φ

definition *AND* **where** *AND* $\Phi \equiv$ *SOME* φ. *fold-graph And-ltlp True-ltlp* Φ φ

lemma *fold-graph-OR*: *finite* $\Phi \Longrightarrow$ *fold-graph Or-ltlp False-ltlp* Φ (*OR* Φ)
unfolding *OR-def* **by** (*rule someI2-ex*[*OF finite-imp-fold-graph*])

lemma *fold-graph-AND*: *finite* $\Phi \Longrightarrow$ *fold-graph And-ltlp True-ltlp* Φ (*AND* Φ)
unfolding *AND-def* **by** (*rule someI2-ex*[*OF finite-imp-fold-graph*])

lemma *holds-of-OR* [*simp*]:
assumes *fin*: *finite* (Φ::'*a pltl set*)
shows ($\sigma \models_p$ *OR* Φ) = ($\exists \varphi \in \Phi$. $\sigma \models_p \varphi$)
proof −
{
fix ψ::'*a pltl*
assume *fold-graph Or-ltlp False-ltlp* Φ ψ
hence ($\sigma \models_p \psi$) = ($\exists \varphi \in \Phi$. $\sigma \models_p \varphi$)
by (*rule fold-graph.induct*) *auto*
}
with *fold-graph-OR*[*OF fin*] **show** *?thesis* **by** *simp*
qed

lemma *holds-of-AND* [*simp*]:
assumes *fin*: *finite* (Φ::'*a pltl set*)
shows ($\sigma \models_p$ *AND* Φ) = ($\forall \varphi \in \Phi$. $\sigma \models_p \varphi$)
proof −
{
fix ψ::'*a pltl*
assume *fold-graph And-ltlp True-ltlp* Φ ψ
hence ($\sigma \models_p \psi$) = ($\forall \varphi \in \Phi$. $\sigma \models_p \varphi$)
by (*rule fold-graph.induct*) *auto*
}
with *fold-graph-AND*[*OF fin*] **show** *?thesis* **by** *simp*
qed

4.2 Next-Free PLTL Formulas

A PLTL formula is called *next-free* if it does not contain any subformula.

fun *next-free* :: '*a pltl* ⇒ *bool*
where
next-free false$_p$ = *True*
| *next-free* (*atom*$_p$(*p*)) = *True*
| *next-free* (φ *implies*$_p$ ψ) = (*next-free* φ ∧ *next-free* ψ)
| *next-free* (X_p φ) = *False*
| *next-free* (φ U_p ψ) = (*next-free* φ ∧ *next-free* ψ)

lemma *next-free-not* [*simp*]:
next-free (*not*$_p$ φ) = *next-free* φ
by (*simp add*: *Not-ltlp-def*)

lemma *next-free-true* [*simp*]:
next-free true$_p$
by (*simp add*: *True-ltlp-def*)

lemma *next-free-or* [*simp*]:

next-free (φ or_p ψ) = (*next-free* φ $\wedge$ *next-free* ψ)
by (*simp add*: *Or-ltlp-def*)

lemma *next-free-and* [*simp*]: *next-free* (φ and_p ψ) = (*next-free* φ $\wedge$ *next-free* ψ)
by (*simp add*: *And-ltlp-def*)

lemma *next-free-eventually* [*simp*]:
next-free (F_p φ) = *next-free* φ
by (*simp add*: *Eventually-ltlp-def*)

lemma *next-free-always* [*simp*]:
next-free (G_p φ) = *next-free* φ
by (*simp add*: *Always-ltlp-def*)

lemma *next-free-release* [*simp*]:
next-free (φ R_p ψ) = (*next-free* φ $\wedge$ *next-free* ψ)
by (*simp add*: *Release-ltlp-def*)

lemma *next-free-weak-until* [*simp*]:
next-free (φ W_p ψ) = (*next-free* φ $\wedge$ *next-free* ψ)
by (*auto simp*: *WeakUntil-ltlp-def*)

lemma *next-free-strong-release* [*simp*]:
next-free (φ M_p ψ) = (*next-free* φ $\wedge$ *next-free* ψ)
by (*auto simp*: *StrongRelease-ltlp-def*)

lemma *next-free-OR* [*simp*]:
assumes *fin*: *finite* (Φ::'*a pltl set*)
shows *next-free* (*OR* Φ) = ($\forall\,\varphi{\in}\Phi$. *next-free* φ)
proof −
{
fix ψ::'*a pltl*
assume *fold-graph Or-ltlp False-ltlp* Φ ψ
hence *next-free* ψ = ($\forall\,\varphi{\in}\Phi$. *next-free* φ)
by (*rule fold-graph.induct*) *auto*
}
with *fold-graph-OR*[*OF fin*] **show** *?thesis* **by** *simp*
qed

lemma *next-free-AND* [*simp*]:
assumes *fin*: *finite* (Φ::'*a pltl set*)
shows *next-free* (*AND* Φ) = ($\forall\,\varphi{\in}\Phi$. *next-free* φ)
proof −
{
fix ψ::'*a pltl*
assume *fold-graph And-ltlp True-ltlp* Φ ψ
hence *next-free* ψ = ($\forall\,\varphi{\in}\Phi$. *next-free* φ)
by (*rule fold-graph.induct*) *auto*
}

with *fold-graph-AND*[*OF fin*] **show** *?thesis* **by** *simp*
qed

4.3 Stuttering Invariance of PLTL Without "Next"

A PLTL formula is *stuttering invariant* if for any stuttering equivalent state sequences $\sigma \approx \tau$, the formula holds of σ iff it holds of τ.

definition *stutter-invariant* **where**
stutter-invariant $\varphi = (\forall \sigma\ \tau.\ (\sigma \approx \tau) \longrightarrow (\sigma \models_p \varphi) = (\tau \models_p \varphi))$

Since stuttering equivalence is symmetric, it is enough to show an implication in the above definition instead of an equivalence.

lemma *stutter-invariantI* [*intro*!]:
assumes $\bigwedge \sigma\ \tau.\ [\![\sigma \approx \tau; \sigma \models_p \varphi]\!] \Longrightarrow \tau \models_p \varphi$
shows *stutter-invariant* φ
proof −
{
fix $\sigma\ \tau$
assume *st*: $\sigma \approx \tau$ **and** *f*: $\sigma \models_p \varphi$
hence $\tau \models_p \varphi$ **by** (*rule assms*)
}
moreover
{
fix $\sigma\ \tau$
assume *st*: $\sigma \approx \tau$ **and** *f*: $\tau \models_p \varphi$
from *st* **have** $\tau \approx \sigma$ **by** (*rule stutter-equiv-sym*)
from *this f* **have** $\sigma \models_p \varphi$ **by** (*rule assms*)
}
ultimately show *?thesis* **by** (*auto simp*: *stutter-invariant-def*)
qed

lemma *stutter-invariantD* [*dest*]:
assumes *stutter-invariant* φ **and** $\sigma \approx \tau$
shows $(\sigma \models_p \varphi) = (\tau \models_p \varphi)$
using *assms* **by** (*auto simp*: *stutter-invariant-def*)

We first show that next-free PLTL formulas are indeed stuttering invariant. The proof proceeds by straightforward induction on the syntax of PLTL formulas.

theorem *next-free-stutter-invariant*:
next-free $\varphi \Longrightarrow$ *stutter-invariant* (φ::'*a pltl*)
proof (*induct* φ)
show *stutter-invariant* $false_p$ **by** *auto*
next
fix $p :: 'a \Rightarrow bool$
show *stutter-invariant* ($atom_p(p)$)
proof
fix $\sigma\ \tau$

```
    assume σ ≈ τ σ ⊨_p atom_p(p)
    thus τ ⊨_p atom_p(p) by (simp add: stutter-equiv-0)
  qed
next
  fix φ ψ :: 'a pltl
  assume ih: next-free φ ⟹ stutter-invariant φ
             next-free ψ ⟹ stutter-invariant ψ
  assume next-free (φ implies_p ψ)
  with ih show stutter-invariant (φ implies_p ψ) by auto
next
  fix φ :: 'a pltl
  assume next-free (X_p φ)  — hence contradiction
  thus stutter-invariant (X_p φ) by simp
next
  fix φ ψ :: 'a pltl
  assume ih: next-free φ ⟹ stutter-invariant φ
             next-free ψ ⟹ stutter-invariant ψ
  assume next-free (φ U_p ψ)
  with ih have stinv: stutter-invariant φ stutter-invariant ψ by auto
  show stutter-invariant (φ U_p ψ)
  proof
    fix σ τ
    assume st: σ ≈ τ and unt: σ ⊨_p φ U_p ψ
    from unt obtain m
      where 1: σ[m..] ⊨_p ψ and 2: ∀ j<m. (σ[j..] ⊨_p φ) by auto
    from st obtain n
      where 3: (σ[m..]) ≈ (τ[n..]) and 4: ∀ i<n. ∃ j<m. (σ[j..]) ≈ (τ[i..])
      by (rule stutter-equiv-suffixes-right)
    from 1 3 stinv have τ[n..] ⊨_p ψ by auto
    moreover
    from 2 4 stinv have ∀ i<n. (τ[i..] ⊨_p φ) by force
    ultimately show τ ⊨_p φ U_p ψ by auto
  qed
qed
```

4.4 Atoms, Canonical State Sequences, and Characteristic Formulas

We now address the converse implication: any stutter invariant PLTL formula φ can be equivalently expressed by a next-free formula. The construction of that formula requires attention to the atomic formulas that appear in φ. We will also prove that the next-free formula does not need any new atoms beyond those present in φ.

The following function collects the atoms (of type $'a \Rightarrow bool$) of a PLTL formula.

```
lemma atoms-pltl-OR [simp]:
  assumes fin: finite (Φ::'a pltl set)
  shows atoms-pltl (OR Φ) = (⋃ φ∈Φ. atoms-pltl φ)
```

proof −
  {
    fix ψ::'*a pltl*
    assume *fold-graph Or-ltlp False-ltlp* Φ ψ
    hence *atoms-pltl* $\psi = (\bigcup \varphi{\in}\Phi.$ *atoms-pltl* $\varphi)$
      by (*rule fold-graph.induct*) *auto*
  }
  with *fold-graph-OR*[*OF fin*] **show** *?thesis* **by** *simp*
qed

lemma *atoms-pltl-AND* [*simp*]:
  assumes *fin*: *finite* (Φ::'*a pltl set*)
  shows *atoms-pltl* (*AND* Φ) = $(\bigcup \varphi{\in}\Phi.$ *atoms-pltl* $\varphi)$
proof −
  {
    fix ψ::'*a pltl*
    assume *fold-graph And-ltlp True-ltlp* Φ ψ
    hence *atoms-pltl* $\psi = (\bigcup \varphi{\in}\Phi.$ *atoms-pltl* $\varphi)$
      by (*rule fold-graph.induct*) *auto*
  }
  with *fold-graph-AND*[*OF fin*] **show** *?thesis* **by** *simp*
qed

Given a set of atoms A as above, we say that two states are A-similar if they agree on all atoms in A. Two state sequences σ and τ are A-similar if corresponding states are A-equal.

definition *state-sim* :: ['*a*, ('*a* ⇒ *bool*) *set*, '*a*] ⇒ *bool*
  (- ~A~ - [70,100,70] 50) **where**
  s ~A~ *t* = (∀ *p*∈*A*. *p s* ⟷ *p t*)

definition *seq-sim* :: [*nat* ⇒ '*a*, ('*a* ⇒ *bool*) *set*, *nat* ⇒ '*a*] ⇒ *bool*
  (- ≃A≃ - [70,100,70] 50) **where**
  $\sigma \simeq A \simeq \tau$ = (∀ *n*. (σ *n*) ~A~ (τ *n*))

These relations are (indexed) equivalence relations. Moreover $s \sim A \sim t$ implies $s \sim B \sim t$ for $B \subseteq A$, and similar for $\sigma \simeq A \simeq \tau$ and $\sigma \simeq B \simeq \tau$.

lemma *state-sim-refl* [*simp*]: *s* ~A~ *s*
  by (*simp add*: *state-sim-def*)

lemma *state-sim-sym*: *s* ~A~ *t* ⟹ *t* ~A~ *s*
  by (*auto simp*: *state-sim-def*)

lemma *state-sim-trans*[*trans*]: *s* ~A~ *t* ⟹ *t* ~A~ *u* ⟹ *s* ~A~ *u*
  unfolding *state-sim-def* **by** *blast*

lemma *state-sim-mono*:
  assumes *s* ~A~ *t* **and** *B* ⊆ *A*
  shows *s* ~B~ *t*
  using *assms* **unfolding** *state-sim-def* **by** *auto*

lemma *seq-sim-refl* [*simp*]: $\sigma \simeq A \simeq \sigma$
by (*simp add*: *seq-sim-def*)

lemma *seq-sim-sym*: $\sigma \simeq A \simeq \tau \Longrightarrow \tau \simeq A \simeq \sigma$
by (*auto simp*: *seq-sim-def state-sim-sym*)

lemma *seq-sim-trans*[*trans*]: $\varrho \simeq A \simeq \sigma \Longrightarrow \sigma \simeq A \simeq \tau \Longrightarrow \varrho \simeq A \simeq \tau$
unfolding *seq-sim-def* **by** (*blast intro*: *state-sim-trans*)

lemma *seq-sim-mono*:
assumes $\sigma \simeq A \simeq \tau$ **and** $B \subseteq A$
shows $\sigma \simeq B \simeq \tau$
using *assms* **unfolding** *seq-sim-def* **by** (*blast intro*: *state-sim-mono*)

State sequences that are similar w.r.t. the atoms of a PLTL formula evaluate that formula to the same value.

lemma *pltl-seq-sim*: $\sigma \simeq \mathit{atoms\text{-}pltl}\ \varphi \simeq \tau \Longrightarrow (\sigma \models_p \varphi) = (\tau \models_p \varphi)$
(**is** *?sim* $\sigma\ \varphi\ \tau \Longrightarrow$ *?P* $\sigma\ \varphi\ \tau$)
proof (*induct* φ *arbitrary*: $\sigma\ \tau$)
fix $\sigma\ \tau$
show *?P* σ false_p τ **by** *simp*
next
fix $p\ \sigma\ \tau$
assume *?sim* σ $(\mathit{atom}_p(p))$ τ **thus** *?P* σ $(\mathit{atom}_p(p))$ τ
by (*auto simp*: *seq-sim-def state-sim-def*)
next
fix $\varphi\ \psi\ \sigma\ \tau$
assume *ih*: $\bigwedge\sigma\ \tau$. *?sim* $\sigma\ \varphi\ \tau \Longrightarrow$ *?P* $\sigma\ \varphi\ \tau$
$\bigwedge\sigma\ \tau$. *?sim* $\sigma\ \psi\ \tau \Longrightarrow$ *?P* $\sigma\ \psi\ \tau$
and *sim*: *?sim* σ $(\varphi\ \mathit{implies}_p\ \psi)$ τ
from *sim* **have** *?sim* $\sigma\ \varphi\ \tau$ *?sim* $\sigma\ \psi\ \tau$
by (*auto elim*: *seq-sim-mono*)
with *ih* **show** *?P* σ $(\varphi\ \mathit{implies}_p\ \psi)$ τ **by** *simp*
next
fix $\varphi\ \sigma\ \tau$
assume *ih*: $\bigwedge\sigma\ \tau$. *?sim* $\sigma\ \varphi\ \tau \Longrightarrow$ *?P* $\sigma\ \varphi\ \tau$
and *sim*: $\sigma \simeq \mathit{atoms\text{-}pltl}\ (X_p\ \varphi) \simeq \tau$
from *sim* **have** $(\sigma[1..]) \simeq \mathit{atoms\text{-}pltl}\ \varphi \simeq (\tau[1..])$
by (*auto simp*: *seq-sim-def*)
with *ih* **show** *?P* σ $(X_p\ \varphi)$ τ **by** *auto*
next
fix $\varphi\ \psi\ \sigma\ \tau$
assume *ih*: $\bigwedge\sigma\ \tau$. *?sim* $\sigma\ \varphi\ \tau \Longrightarrow$ *?P* $\sigma\ \varphi\ \tau$
$\bigwedge\sigma\ \tau$. *?sim* $\sigma\ \psi\ \tau \Longrightarrow$ *?P* $\sigma\ \psi\ \tau$
and *sim*: *?sim* σ $(\varphi\ U_p\ \psi)$ τ
from *sim* **have** $\forall\, i.\ (\sigma[i..]) \simeq \mathit{atoms\text{-}pltl}\ \varphi \simeq (\tau[i..])\ \forall\, j.\ (\sigma[j..]) \simeq \mathit{atoms\text{-}pltl}\ \psi \simeq (\tau[j..])$
by (*auto simp*: *seq-sim-def state-sim-def*)

with *ih* **have** $\forall\, i.\ ?P\ (\sigma[i..])\ \varphi\ (\tau[i..])\ \forall\, j.\ ?P\ (\sigma[j..])\ \psi\ (\tau[j..])$
by *blast+*
thus $?P\ \sigma\ (\varphi\ U_p\ \psi)\ \tau$
by (*meson semantics-pltl.simps*(5))
qed

The following function picks an arbitrary representative among A-similar states. Because the choice is functional, any two A-similar states are mapped to the same state.

definition *canonize* **where**
$\mathit{canonize}\ A\ s \equiv \mathit{SOME}\ t.\ t\ {\sim}A{\sim}\ s$

lemma *canonize-state-sim*: $\mathit{canonize}\ A\ s\ {\sim}A{\sim}\ s$
unfolding *canonize-def* **by** (*rule someI*, *rule state-sim-refl*)

lemma *canonize-canonical*:
assumes *st*: $s\ {\sim}A{\sim}\ t$
shows $\mathit{canonize}\ A\ s = \mathit{canonize}\ A\ t$
proof −
from *st* **have** $\forall\, u.\ (u\ {\sim}A{\sim}s) = (u\ {\sim}A{\sim}\ t)$
by (*auto elim*: *state-sim-sym state-sim-trans*)
thus *?thesis* **unfolding** *canonize-def* **by** *simp*
qed

lemma *canonize-idempotent*:
$\mathit{canonize}\ A\ (\mathit{canonize}\ A\ s) = \mathit{canonize}\ A\ s$
by (*rule canonize-canonical*[*OF canonize-state-sim*])

In a canonical state sequence, any two A-similar states are in fact equal.

definition *canonical-sequence* **where**
$\mathit{canonical\text{-}sequence}\ A\ \sigma \equiv \forall\, m\ (n{::}\mathit{nat}).\ \sigma\ m\ {\sim}A{\sim}\ \sigma\ n \longrightarrow \sigma\ m = \sigma\ n$

Every suffix of a canonical sequence is canonical, as is any (sampled) subsequence, in particular any stutter-sampling.

lemma *canonical-suffix*:
$\mathit{canonical\text{-}sequence}\ A\ \sigma \Longrightarrow \mathit{canonical\text{-}sequence}\ A\ (\sigma[k..])$
by (*auto simp*: *canonical-sequence-def*)

lemma *canonical-sampled*:
$\mathit{canonical\text{-}sequence}\ A\ \sigma \Longrightarrow \mathit{canonical\text{-}sequence}\ A\ (\sigma \circ f)$
by (*auto simp*: *canonical-sequence-def*)

lemma *canonical-reduced*:
$\mathit{canonical\text{-}sequence}\ A\ \sigma \Longrightarrow \mathit{canonical\text{-}sequence}\ A\ (\natural\sigma)$
unfolding *stutter-reduced-def* **by** (*rule canonical-sampled*)

For any sequence σ there exists a canonical A-similar sequence τ. Such a τ can be obtained by canonizing all states of σ.

lemma *canonical-exists*:
obtains τ **where** $\tau \simeq A \simeq \sigma$ *canonical-sequence* A τ
proof −
have (*canonize* $A \circ \sigma$) $\simeq A \simeq \sigma$
by (*simp add*: *seq-sim-def canonize-state-sim*)
moreover
have *canonical-sequence* A (*canonize* $A \circ \sigma$)
by (*auto simp*: *canonical-sequence-def canonize-idempotent*
dest: *canonize-canonical*)
ultimately
show *?thesis* **using** *that* **by** *blast*
qed

Given a state s and a set A of atoms, we define the characteristic formula of s as the conjunction of all atoms in A that hold of s and the negation of the atoms in A that do not hold of s.

definition *characteristic-formula* **where**
characteristic-formula A $s \equiv$
$((AND\ \{\ atom_p(p) \mid p\ .\ p \in A \wedge p\ s\ \})\ and_p\ (AND\ \{\ not_p\ (atom_p(p)) \mid p\ .\ p \in A \wedge \neg(p\ s)\ \}))$

lemma *characteristic-holds*:
finite $A \Longrightarrow \sigma \models_p$ *characteristic-formula* A $(\sigma\ 0)$
by (*auto simp*: *characteristic-formula-def*)

lemma *characteristic-state-sim*:
assumes *fin*: *finite* A
shows $(\sigma\ 0\ {\sim}A{\sim}\ \tau\ 0) = (\tau \models_p$ *characteristic-formula* A $(\sigma\ (0{::}nat)))$
proof
assume *sim*: $\sigma\ 0\ {\sim}A{\sim}\ \tau\ 0$
{
fix p
assume $p \in A$
with *sim* **have** $p\ (\tau\ 0) = p\ (\sigma\ 0)$ **by** (*auto simp*: *state-sim-def*)
}
with *fin* **show** $\tau \models_p$ *characteristic-formula* A $(\sigma\ 0)$
by (*auto simp*: *characteristic-formula-def*) (*blast*+)
next
assume $\tau \models_p$ *characteristic-formula* A $(\sigma\ 0)$
with *fin* **show** $\sigma\ 0\ {\sim}A{\sim}\ \tau\ 0$
by (*auto simp*: *characteristic-formula-def state-sim-def*)
qed

4.5 Stuttering Invariant PLTL Formulas Don't Need Next

The following is the main lemma used in the proof of the completeness theorem: for any PLTL formula φ there exists a next-free formula ψ such that the two formulas evaluate to the same value over stutter-free and canonical

sequences (w.r.t. some $A \supseteq$ *atoms-pltl* φ).

lemma *ex-next-free-stutter-free-canonical*:
  assumes A: *atoms-pltl* $\varphi \subseteq A$ **and** *fin*: *finite* A
  shows $\exists \psi$. *next-free* $\psi \wedge$ *atoms-pltl* $\psi \subseteq A \wedge$
        $(\forall \sigma$. *stutter-free* $\sigma \wedge$ *canonical-sequence* $A\ \sigma \longrightarrow (\sigma \models_p \psi) = (\sigma \models_p \varphi))$
    (**is** $\exists \psi$. *?P* $\varphi\ \psi$)
using A **proof** (*induct* φ)

The cases of *false* and atomic formulas are trivial.

  have *?P* $false_p$ $false_p$ **by** *auto*
  thus $\exists \psi$. *?P* $false_p\ \psi$ **..**
next
  fix p
  assume *atoms-pltl* $(atom_p(p)) \subseteq A$
  hence *?P* $(atom_p(p))$ $(atom_p(p))$ **by** *auto*
  thus $\exists \psi$. *?P* $(atom_p(p))\ \psi$ **..**
next

Implication is easy, using the induction hypothesis.

  fix $\varphi\ \psi$
  assume *atoms-pltl* $\varphi \subseteq A \Longrightarrow \exists \varphi'$. *?P* $\varphi\ \varphi'$
    and *atoms-pltl* $\psi \subseteq A \Longrightarrow \exists \psi'$. *?P* $\psi\ \psi'$
    and *atoms-pltl* $(\varphi\ implies_p\ \psi) \subseteq A$
  then obtain $\varphi'\ \psi'$ **where** *?P* $\varphi\ \varphi'$ *?P* $\psi\ \psi'$ **by** *auto*
  hence *?P* $(\varphi\ implies_p\ \psi)$ $(\varphi'\ implies_p\ \psi')$ **by** *auto*
  thus $\exists \chi$. *?P* $(\varphi\ implies_p\ \psi)\ \chi$ **..**
next

The case of *until* follows similarly.

  fix $\varphi\ \psi$
  assume *atoms-pltl* $\varphi \subseteq A \Longrightarrow \exists \varphi'$. *?P* $\varphi\ \varphi'$
    and *atoms-pltl* $\psi \subseteq A \Longrightarrow \exists \psi'$. *?P* $\psi\ \psi'$
    and *atoms-pltl* $(\varphi\ U_p\ \psi) \subseteq A$
  then obtain $\varphi'\ \psi'$ **where** *1*: *?P* $\varphi\ \varphi'$ **and** *2*: *?P* $\psi\ \psi'$ **by** *auto*
  {
    fix σ
    assume *sigma*: *stutter-free* σ *canonical-sequence* $A\ \sigma$
    hence $\bigwedge k$. *stutter-free* $(\sigma[k..])$ $\bigwedge k$. *canonical-sequence* A $(\sigma[k..])$
      by (*auto simp*: *stutter-free-suffix canonical-suffix*)
    with *1 2*
    have $\bigwedge k$. $(\sigma[k..] \models_p \varphi') = (\sigma[k..] \models_p \varphi)$
     and $\bigwedge k$. $(\sigma[k..] \models_p \psi') = (\sigma[k..] \models_p \psi)$
      by (*blast+*)
    hence $(\sigma \models_p \varphi'\ U_p\ \psi') = (\sigma \models_p \varphi\ U_p\ \psi)$
      by *auto*
  }
  with *1 2* **have** *?P* $(\varphi\ U_p\ \psi)$ $(\varphi'\ U_p\ \psi')$ **by** *auto*
  thus $\exists \chi$. *?P* $(\varphi\ U_p\ \psi)\ \chi$ **..**
next

The interesting case is the one of the *next*-operator.

fix φ
assume *ih*: *atoms-pltl* $\varphi \subseteq A \Longrightarrow \exists \psi.$ *?P* φ ψ **and** *at*: *atoms-pltl* $(X_p\ \varphi) \subseteq A$
then obtain ψ **where** *psi*: *?P* φ ψ **by** *auto*

A valuation (over A) is a set $val \subseteq A$ of atoms. We define some auxiliary notions: the valuation corresponding to a state and the characteristic formula for a valuation. Finally, we define the formula *psi'* that we will prove to be equivalent to $X_p\ \varphi$ over the stutter-free and canonical sequence σ.

define *stval* **where** *stval* $= (\lambda s.\ \{\ p \in A\ .\ p\ s\ \})$
define *chi* **where** *chi* $= (\lambda val.\ ((AND\ \{atom_p(p) \mid p\ .\ p \in val\})\ and_p$
$(AND\ \{not_p\ (atom_p(p)) \mid p\ .\ p \in A - val\})))$
define *psi'* **where** *psi'* $= ((\psi\ and_p\ (OR\ \{G_p\ (chi\ val) \mid val\ .\ val \subseteq A\ \}))\ or_p$
$(OR\ \{(chi\ val)\ and_p\ ((chi\ val)\ U_p\ (\ \psi\ and_p\ (chi\ val'))) \mid val\ val'.$
$val \subseteq A \wedge val' \subseteq A \wedge val' \neq val\ \}))$
(**is** - $= ((\ -\ and_p\ (OR\ ?ALW))\ or_p\ (OR\ ?UNT)))$

have $\bigwedge s.\ \{not_p\ (atom_p(p)) \mid p\ .\ p \in A - stval\ s\}$
$= \{not_p\ (atom_p(p)) \mid p\ .\ p \in A \wedge \neg(p\ s)\}$
by (*auto simp*: *stval-def*)
hence *chi1*: $\bigwedge s.$ *chi* $(stval\ s) =$ *characteristic-formula A s*
by (*auto simp*: *chi-def stval-def characteristic-formula-def*)
{
fix *val* τ
assume *val*: $val \subseteq A$ **and** *tau*: $\tau \models_p$ *chi val*
with *fin* **have** *val* $=$ *stval* $(\tau\ 0)$
by (*auto simp*: *stval-def chi-def finite-subset*)
}
note *chi2* $=$ *this*

have *?UNT* $\subseteq (\lambda(val,val').\ (chi\ val)\ and_p\ ((chi\ val)\ U_p\ (\psi\ and_p\ (chi\ val'))))$
$`\ (Pow\ A \times Pow\ A)$
(**is** - $\subseteq$ *?S*)
by *auto*
with *fin* **have** *fin-UNT*: *finite ?UNT*
by (*auto simp*: *finite-subset*)

have *nf*: *next-free psi'*
proof −
from *fin* **have** $\bigwedge val.\ val \subseteq A \Longrightarrow$ *next-free* $(chi\ val)$
by (*auto simp*: *chi-def finite-subset*)
with *fin fin-UNT psi* **show** *?thesis*
by (*force simp*: *psi'-def finite-subset*)
qed

have *atoms-pltl*: *atoms-pltl psi'* $\subseteq A$
proof −
from *fin* **have** *at-chi*: $\bigwedge val.\ val \subseteq A \Longrightarrow$ *atoms-pltl* $(chi\ val) \subseteq A$
by (*auto simp*: *chi-def finite-subset*)

```
  with fin psi have at-alw: atoms-pltl (ψ and_p (OR ?ALW)) ⊆ A
    by auto blast?
  from fin fin-UNT psi at-chi have atoms-pltl (OR ?UNT) ⊆ A
    by auto (blast+)?
  with at-alw show ?thesis by (auto simp: psi'-def)
qed

{
  fix σ
  assume st: stutter-free σ and can: canonical-sequence A σ
  have (σ ⊨_p X_p φ) = (σ ⊨_p psi')
  proof (cases σ (Suc 0) = σ 0)
    case True
```

In the case of a stuttering transition at the beginning, we must have infinite stuttering, and the first disjunct of *psi'* holds, whereas the second does not.

```
    {
      fix n
      have σ n = σ 0
      proof (cases n)
        case 0 thus ?thesis by simp
      next
        case Suc
        hence n > 0 by simp
        with True st show ?thesis unfolding stutter-free-def by blast
      qed
    }
    note alleq = this
    have suffix: ⋀n. σ[n..] = σ
    proof (rule ext)
      fix n i
      have (σ[n..]) i = σ 0 by (auto intro: alleq)
      moreover have σ i = σ 0 by (rule alleq)
      ultimately show (σ[n..]) i = σ i by simp
    qed
    with st can psi have 1: (σ ⊨_p X_p φ) = (σ ⊨_p ψ) by simp

   from fin have σ ⊨_p chi (stval (σ 0)) by (simp add: chi1 characteristic-holds)
    with suffix have σ ⊨_p G_p (chi (stval (σ 0))) (is - ⊨_p ?alw) by simp
    moreover have ?alw ∈ ?ALW by (auto simp: stval-def)
    ultimately have 2: σ ⊨_p OR ?ALW
      using fin by (auto simp: finite-subset simp del: semantics-pltl-sugar)

    have 3: ¬(σ ⊨_p OR ?UNT)
    proof
      assume unt: σ ⊨_p OR ?UNT
      with fin-UNT obtain val val' k where
        val: val ⊆ A val' ⊆ A val' ≠ val and
        now: σ ⊨_p chi val and k: σ[k..] ⊨_p chi val'
```

```
      by auto (blast+)?
    from ‹val ⊆ A› now have val = stval (σ 0) by (rule chi2)
    moreover
    from ‹val' ⊆ A› k suffix have val' = stval (σ 0) by (simp add: chi2)
    moreover note ‹val' ≠ val›
    ultimately show False by simp
  qed

  from 1 2 3 show ?thesis by (simp add: psi'-def)

next
  case False
```

Otherwise, $\sigma \models_p X_p\ \varphi$ is equivalent to σ satisfying the second disjunct of *psi'*. We show both implications separately.

```
  let ?val = stval (σ 0)
  let ?val' = stval (σ 1)
  from False can have vals: ?val' ≠ ?val
    by (auto simp: canonical-sequence-def state-sim-def stval-def)

  show ?thesis
  proof
    assume phi: σ ⊨p X_p φ
    from fin have 1: σ ⊨p chi ?val by (simp add: chi1 characteristic-holds)

    from st can have stutter-free (σ[1..]) canonical-sequence A (σ[1..])
      by (auto simp: stutter-free-suffix canonical-suffix)
    with phi psi have 2: σ[1..] ⊨p ψ by auto

    from fin have σ[1..] ⊨p characteristic-formula A ((σ[1..]) 0)
      by (rule characteristic-holds)
    hence 3: σ[1..] ⊨p chi ?val' by (simp add: chi1)

    from 1 2 3 have σ ⊨p And-ltlp (chi ?val) ((chi ?val) U_p (And-ltlp ψ (chi
?val')))
      (is - ⊨p ?unt)
      by auto
    moreover from vals have ?unt ∈ ?UNT
      by (auto simp: stval-def)
    ultimately have σ ⊨p OR ?UNT
      using fin-UNT[THEN holds-of-OR] by blast
    thus σ ⊨p psi' by (simp add: psi'-def)

  next
    assume psi': σ ⊨p psi'
    have ¬(σ ⊨p OR ?ALW)
    proof
      assume σ ⊨p OR ?ALW
      with fin obtain val where 1: val ⊆ A and 2: ∀ n. (σ[n..] ⊨p chi val)
```

```
      by (force simp: finite-subset)
    from 2 have σ[0..] ⊨p chi val ..
    with 1 have val = ?val by (simp add: chi2)
    moreover
    from 2 have σ[1..] ⊨p chi val ..
    with 1 have val = ?val' by (force dest: chi2)
    ultimately
    show False using vals by simp
  qed
  with psi' have σ ⊨p OR ?UNT by (simp add: psi'-def)
  with fin-UNT obtain val val' k where
    val: val ⊆ A val' ⊆ A val' ≠ val and
    now: σ ⊨p chi val and
    k: σ[k..] ⊨p ψ σ[k..] ⊨p chi val' and
    i: ∀ i<k. (σ[i..] ⊨p chi val)
    by auto (blast+)?

  from val now have 1: val = ?val by (simp add: chi2)

  have 2: k ≠ 0
  proof
    assume k=0
    with val k have val' = ?val by (simp add: chi2)
    with 1 ‹val' ≠ val› show False by simp
  qed

  have 3: k ≤ 1
  proof (rule ccontr)
    assume ¬(k ≤ 1)
    with i have σ[1..] ⊨p chi val by simp
    with 1 have σ[1..] ⊨p characteristic-formula A (σ 0)
      by (simp add: chi1)
    hence (σ 0) ~A~ ((σ[1..]) 0)
      using characteristic-state-sim[OF fin] by blast
    with can have σ 0 = σ 1
      by (simp add: canonical-sequence-def)
    with ‹σ (Suc 0) ≠ σ 0› show False by simp
  qed

  from 2 3 have k=1 by simp
  moreover
  from st can have stutter-free (σ[1..]) canonical-sequence A (σ[1..])
    by (auto simp: stutter-free-suffix canonical-suffix)
  ultimately show σ ⊨p X_p φ using ‹σ[k..] ⊨p ψ› psi by auto
 qed
 qed
 }
 with nf atoms-pltl show ∃ψ'. ?P (X_p φ) ψ' by blast
qed
```

Comparing the definition of the next-free formula in the case of formulas $X_p\ \varphi$ with the one that appears in [2], there is a subtle difference. Peled and Wilke define the second disjunct as a disjunction of formulas

$$(chi\ val)\ U_p\ (\psi\ and_p\ (chi\ val'))$$

for subsets $val, val' \subseteq A$ whereas we conjoin the formula *chi val* to the "until" formula. This conjunct is indeed necessary in order to rule out the case of the "until" formula being true because of *chi val'* being true immediately. The subtle error in the definition of the formula was acknowledged by Peled and Wilke and apparently had not been noticed since the publication of [2] in 1996 (which has been cited more than a hundred times according to Google Scholar). Although the error was corrected easily, the fact that authors, reviewers, and readers appear to have missed it for so long underscores the usefulness of formal proofs.

We now show that any stuttering invariant PLTL formula can be expressed without the X_p operator.

theorem *stutter-invariant-next-free*:
assumes *phi*: *stutter-invariant* φ
obtains ψ **where** *next-free* ψ *atoms-pltl* $\psi \subseteq$ *atoms-pltl* φ
$\forall \sigma.\ (\sigma \models_p \psi) = (\sigma \models_p \varphi)$
proof −
have *atoms-pltl* $\varphi \subseteq$ *atoms-pltl* φ *finite* (*atoms-pltl* φ) **by** *simp-all*
then obtain ψ **where**
psi: *next-free* ψ *atoms-pltl* $\psi \subseteq$ *atoms-pltl* φ **and**
equiv: $\forall \sigma.$ *stutter-free* $\sigma \wedge$ *canonical-sequence* (*atoms-pltl* φ) $\sigma \longrightarrow (\sigma \models_p \psi) = (\sigma \models_p \varphi)$
by (*blast dest*: *ex-next-free-stutter-free-canonical*)
from ‹*next-free* ψ› **have** *sinv*: *stutter-invariant* ψ
by (*rule next-free-stutter-invariant*)
{
fix σ
obtain τ **where** *1*: $\tau \simeq$ *atoms-pltl* $\varphi \simeq \sigma$ **and** *2*: *canonical-sequence* (*atoms-pltl* φ) τ
by (*rule canonical-exists*)
from *1* ‹*atoms-pltl* $\psi \subseteq$ *atoms-pltl* φ› **have** *3*: $\tau \simeq$ *atoms-pltl* $\psi \simeq \sigma$
by (*rule seq-sim-mono*)

from *1* **have** $(\sigma \models_p \varphi) = (\tau \models_p \varphi)$ **by** (*simp add*: *pltl-seq-sim*)
also from *phi stutter-reduced-equivalent* **have** ... = $(\natural\tau \models_p \varphi)$ **by** *auto*
also from *2*[*THEN canonical-reduced*] *equiv stutter-reduced-stutter-free*
have ... = $(\natural\tau \models_p \psi)$ **by** *auto*
also from *sinv stutter-reduced-equivalent* **have** ... = $(\tau \models_p \psi)$ **by** *auto*
also from *3* **have** ... = $(\sigma \models_p \psi)$ **by** (*simp add*: *pltl-seq-sim*)
finally have $(\sigma \models_p \psi) = (\sigma \models_p \varphi)$ **by** (*rule sym*)
}
with *psi that* **show** *?thesis* **by** *blast*

qed

Combining theorems *next-free-stutter-invariant* and *stutter-invariant-next-free*, it follows that a PLTL formula is stuttering invariant iff it is equivalent to a next-free formula.

theorem *pltl-stutter-invariant*:
stutter-invariant $\varphi \longleftrightarrow$
($\exists \psi$. *next-free* $\psi \wedge$ *atoms-pltl* $\psi \subseteq$ *atoms-pltl* $\varphi \wedge (\forall \sigma.\ \sigma \models_p \psi \longleftrightarrow \sigma \models_p \varphi)$)
proof −
{
assume *stutter-invariant* φ
hence $\exists \psi$. *next-free* $\psi \wedge$ *atoms-pltl* $\psi \subseteq$ *atoms-pltl* $\varphi \wedge (\forall \sigma.\ \sigma \models_p \psi \longleftrightarrow \sigma \models_p \varphi)$
by (*rule stutter-invariant-next-free*) *blast*
}
moreover
{
fix ψ
assume *1*: *next-free* ψ **and** *2*: $\forall \sigma.\ \sigma \models_p \psi \longleftrightarrow \sigma \models_p \varphi$
from *1* **have** *stutter-invariant* ψ **by** (*rule next-free-stutter-invariant*)
with *2* **have** *stutter-invariant* φ **by** *blast*
}
ultimately show *?thesis* **by** *blast*
qed

4.6 Stutter Invariance for LTL with Syntactic Sugar

We lift the results for PLTL to an extensive version of LTL.

primrec *ltlc-next-free* :: *'a ltlc* ⇒ *bool*
where
ltlc-next-free $true_c$ = *True*
| *ltlc-next-free* $false_c$ = *True*
| *ltlc-next-free* ($prop_c(q)$) = *True*
| *ltlc-next-free* ($not_c\ \varphi$) = *ltlc-next-free* φ
| *ltlc-next-free* ($\varphi\ and_c\ \psi$) = (*ltlc-next-free* $\varphi \wedge$ *ltlc-next-free* ψ)
| *ltlc-next-free* ($\varphi\ or_c\ \psi$) = (*ltlc-next-free* $\varphi \wedge$ *ltlc-next-free* ψ)
| *ltlc-next-free* ($\varphi\ implies_c\ \psi$) = (*ltlc-next-free* $\varphi \wedge$ *ltlc-next-free* ψ)
| *ltlc-next-free* ($X_c\ \varphi$) = *False*
| *ltlc-next-free* ($F_c\ \varphi$) = *ltlc-next-free* φ
| *ltlc-next-free* ($G_c\ \varphi$) = *ltlc-next-free* φ
| *ltlc-next-free* ($\varphi\ U_c\ \psi$) = (*ltlc-next-free* $\varphi \wedge$ *ltlc-next-free* ψ)
| *ltlc-next-free* ($\varphi\ R_c\ \psi$) = (*ltlc-next-free* $\varphi \wedge$ *ltlc-next-free* ψ)
| *ltlc-next-free* ($\varphi\ W_c\ \psi$) = (*ltlc-next-free* $\varphi \wedge$ *ltlc-next-free* ψ)
| *ltlc-next-free* ($\varphi\ M_c\ \psi$) = (*ltlc-next-free* $\varphi \wedge$ *ltlc-next-free* ψ)

lemma *ltlc-next-free-iff*[*simp*]: *next-free* (*ltlc-to-pltl* φ) $\longleftrightarrow$ *ltlc-next-free* φ
by (*induction* φ) *auto*

A next free formula cannot distinguish between stutter-equivalent runs.

theorem *ltlc-next-free-stutter-invariant*:
  assumes *next-free*: *ltlc-next-free* φ
  assumes *eq*: $r \approx r'$
  shows $r \models_c \varphi \longleftrightarrow r' \models_c \varphi$
proof −
  {
    fix $r\ r'$
    assume *eq*: $r \approx r'$ **and** *holds*: $r \models_c \varphi$
    then have $r \models_p$ (*ltlc-to-pltl* φ)**by** *simp*

    from *next-free-stutter-invariant*[*of ltlc-to-pltl* φ] *next-free*
    have *PLTL.stutter-invariant* (*ltlc-to-pltl* φ) **by** *simp*
    from *stutter-invariantD*[*OF this eq*] *holds* **have** $r' \models_c \varphi$ **by** *simp*
  } **note** *aux*=*this*

  from *aux*[*of r r'*] *aux*[*of r' r*] *eq stutter-equiv-sym*[*OF eq*] **show** *?thesis*
    by *blast*
qed

end

References

[1] L. Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 657–668, Paris, Sept. 1983. IFIP, North-Holland.

[2] D. Peled and T. Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Inf. Proc. Lett.*, 63(5):243–246, 1997.