# LinguaFrancaClocks

Frédéric Boulanger

March 6, 2020

## Contents



**theory** *LinguaFrancaClocks*

**imports** *Main*

**begin**

## 1 Basic definitions

Instants are represented as the natural numbers. A clock represents an event that may occur or not at any instant. We model a clock as a function from nat to bool, which is True at every instant when the clock ticks (the event occurs).

**type-synonym** *clock* = ‹*nat* ⇒ *bool*›

## 1.1 Periodic clocks

A clock is (k,p)-periodic if it ticks at instants separated by p instants, starting at instant k.

**definition** *kp-periodic* :: ‹[*nat*, *nat*, *clock*] ⇒ *bool*›
  **where** ‹*kp-periodic k p c* ≡
    $(p > 0) \wedge (\forall n.\ c\ n = ((n \geq k) \wedge ((n - k)\ mod\ p = 0)))$›

A 1-periodic clock always ticks starting at its offset

**lemma** *one-periodic-ticks*:
  **assumes** ‹*kp-periodic k 1 c*›
    **and** ‹$n \geq k$›
  **shows** ‹*c n*›
**using** *assms kp-periodic-def* **by** *simp*

A p-periodic clock is a (k,p)-periodic clock starting from a given offset.

**definition** ‹*p-periodic p c* ≡ ($\exists k$. *kp-periodic k p c*)›

**lemma** *p-periodic-intro*[*intro*]:
  ‹*kp-periodic k p c* ⟹ *p-periodic p c*›
**using** *p-periodic-def* **by** *blast*

No clock is 0-periodic.

**lemma** *no-0-periodic*:
  ‹¬*p-periodic 0 c*›
**by** (*simp add*: *kp-periodic-def p-periodic-def*)

A periodic clock is a p-periodic clock for a given period.

**definition** ‹*periodic c* ≡ ($\exists p$. *p-periodic p c*)›

**lemma** *periodic-intro1*[*intro*]:
  ‹*p-periodic p c* ⟹ *periodic c*›
**using** *p-periodic-def periodic-def* **by** *blast*

**lemma** *periodic-intro2*[*intro*]:
  ‹*kp-periodic k p c* ⟹ *periodic c*›
**using** *p-periodic-intro periodic-intro1* **by** *blast*

## 1.2 Sporadic clocks

A clock is p-sporadic if it ticks at instants separated at least by p instants.

**definition** *p-sporadic* :: ‹[*nat*, *clock*] ⇒ *bool*›
  **where** ‹*p-sporadic p c* ≡ $\forall t.\ c\ t \longrightarrow (\forall t'.\ (t' > t \wedge c\ t') \longrightarrow t' > t + p)$›

Any clock is 0-sporadic

**lemma** *sporadic-0*: ‹*p-sporadic 0 c*›
  **unfolding** *p-sporadic-def* **by** *auto*

We define sporadic clock as p-sporadic clocks for some non null interval p.

**definition** ‹*sporadic c* ≡ (∃ *p* > *0*. *p-sporadic p c*)›

**lemma** *sporadic-intro*[*intro*]
  :‹⟦*p-sporadic p c*;*p* > *0*⟧ ⟹ *sporadic c*›
**using** *sporadic-def* **by** *blast*

# 2 Properties of clocks

Some useful lemmas about modulo.

**lemma** *mod-sporadic*:
  **assumes** ‹((*n*::*nat*) *mod p* = *0*)›
    **shows** ‹∀ *n*′. (*n* < *n*′ ∧ *n*′ < *n*+*p*) ⟶ ¬(*n*′ *mod p* = *0*)›
**using** *assms less-imp-add-positive* **by** *fastforce*

**lemma** *mod-sporadic*′:
  **assumes** ‹((*n*::*nat*) *mod p* = *0*)›
    **shows** ‹∀ *n*′. (*n* < *n*′ ∧ (*n*′ *mod p* = *0*)) ⟶ *n*′ ≥ *n*+*p*›
**proof** −
  **{ fix** *n*′ **assume** *h*:‹*n* < *n*′ ∧ *n*′ *mod p* = *0*›
    **hence** ‹*n*′ ≥ *n*+*p*› **using** *mod-sporadic*[*OF assms*] **by** *auto*
  **} thus** *?thesis* **by** *simp*
**qed**

**lemma** *mod-offset-sporadic*:
  **assumes** ‹(*n*::*nat*) ≥ *k*›
      **and** ‹(*n* − *k*) *mod p* = *0*›
    **shows** ‹∀ *n*′. (*n* < *n*′ ∧ *n*′ < *n*+*p*) ⟶ ¬((*n*′−*k*) *mod p* = *0*)›
**proof** −
 **from** *assms* **have** ‹∀ *n*′. *n*′ > *n* ⟶ (*n*′−*k*) > (*n*−*k*)› **by** (*simp add*: *diff-less-mono*)
  **with** *mod-sporadic*[*OF assms*(*2*)] **show** *?thesis* **by** *auto*
**qed**

**lemma** *mod-offset-sporadic*′:
  **assumes** ‹(*n*::*nat*) ≥ *k*›
      **and** ‹(*n* − *k*) *mod p* = *0*›
    **shows** ‹∀ *n*′. (*n* < *n*′ ∧ ((*n*′−*k*) *mod p* = *0*)) ⟶ *n*′ ≥ *n*+*p*›
**proof** −
 **from** *assms* **have** ‹∀ *n*′. *n*′ > *n* ⟶ (*n*′−*k*) > (*n*−*k*)› **by** (*simp add*: *diff-less-mono*)
  **with** *mod-sporadic*[*OF assms*(*2*)] **show** *?thesis* **by** *auto*
**qed**

A (p+1)-periodic clock is p-sporadic.

**lemma** *periodic-suc-sporadic*:
  **assumes** ‹*p-periodic* (*p*+*1*) *c*›
    **shows** ‹*p-sporadic p c*›
**proof** −

**from** *assms p-periodic-def* **obtain** *k*
  **where** ‹*kp-periodic k* (*Suc p*) *c*› **by** (*auto simp add*: *Suc-eq-plus1*[*symmetric*])
  **hence** ‹$\forall n.\ c\ n = ((n \geq k) \wedge ((n - k)\ mod\ (Suc\ p) = 0))$› **unfolding** *kp-periodic-def* **by** *simp*
**thus** *?thesis*
  **unfolding** *p-sporadic-def*
  **using** *mod-offset-sporadic'*[**where** *k*=*k* **and** *p*=‹*Suc p*›]
  **by** (*simp add*: *Suc-le-lessD*)
**qed**

# 3 Operations on clocks

The result of merging two clocks ticks whenever any of the two clocks ticks.

**definition** *merge* :: ‹[*clock*, *clock*] ⇒ *clock*› (**infix** ‹⊕› *60*)
  **where** ‹$c1 \oplus c2 \equiv \lambda t.\ c1\ t \vee c2\ t$›

**lemma** *merge-comm*: ‹$c \oplus c' = c' \oplus c$›
**by** (*auto simp add*: *merge-def*)

Delaying a clock by one instant.

**definition** *delay* :: ‹*clock* ⇒ *clock*› (‹$›)
  **where** ‹$c k = (*case k of 0* ⇒ *False* | *Suc k'* ⇒ *c k'*)›

Sampling a clock with another clock.

**definition** *sampling* :: ‹[*clock*, *clock*] ⇒ *clock*› (**infix** ‹*when*› *70*)
  **where** ‹$c\ when\ c' \equiv \lambda k.\ c\ k \wedge c'\ k$›

**lemma** *sampling-comm*: ‹$c\ when\ c' = c'\ when\ c$›
**by** (*auto simp add*: *sampling-def*)

Merging two sporadic clocks does not necessary yields a sporadic clock.

**lemma** *merge-no-sporadic*:
  ‹$\exists\, c\ c'.\ sporadic\ c \wedge sporadic\ c' \wedge \neg sporadic\ (c \oplus c')$›
**proof** −
  **define** *c* :: *clock* **where** ‹$c = (\lambda t.\ t\ mod\ 2 = 0)$›
  **define** *c'* :: *clock* **where** ‹$c' = (\lambda t.\ t \geq 1 \wedge (t-1)\ mod\ 2 = 0)$›

  **have** ‹*p-periodic 2 c*› **unfolding** *p-periodic-def kp-periodic-def*
        **using** *c-def* **by** *auto*
  **hence** *1*:‹*sporadic c*›
    **using** *periodic-suc-sporadic Suc-1*[*symmetric*] *sporadic-def zero-less-one*
    **by** *auto*

  **have** ‹*p-periodic 2 c'*› **unfolding** *p-periodic-def kp-periodic-def* **using** *c'-def*
    **by** *auto*
  **hence** *2*:‹*sporadic c'*›
    **using** *periodic-suc-sporadic Suc-1*[*symmetric*] *sporadic-def zero-less-one*

**by** *auto*

**have** ‹$\neg$*sporadic* $(c \oplus c')$›
**proof** −
{ **assume** ‹*sporadic* $(c \oplus c')$›
**from** *this* **obtain** $p$ **where** $*$:‹$p > 0$› **and** ‹*p-sporadic* $p$ $(c \oplus c')$›
**using** *sporadic-def* **by** *blast*
**hence** ‹$\forall t.\ (c \oplus c')\ t \longrightarrow (\forall t'.\ (t < t' \land (c \oplus c') t') \longrightarrow\ t' > t+p)$›
**by** (*simp add*:*p-sporadic-def*)
**moreover have** ‹$(c \oplus c')\ 0$› **using** *c-def c′-def merge-def* **by** *simp*
**moreover have** ‹$(c \oplus c')\ 1$› **using** *c-def c′-def merge-def* **by** *simp*
**ultimately have** *False* **using** $*$ **by** *blast*
} **thus** *?thesis* **..**
**qed**
**with** *1* **and** *2* **show** *?thesis* **by** *blast*
**qed**

Delaying a periodic clock yields a shifted periodic clock.

**lemma** *delay-shift-periodic*:
**assumes** ‹*kp-periodic* $k$ $p$ $c$›
**shows** ‹*kp-periodic* $(k+1)$ $p$ $(\$c)$›
**proof** −
**from** *assms* **have** *1*:‹$(p > 0)$› **and** *2*:‹$(\forall n.\ c\ n = ((n \geq k) \land ((n - k)\ mod\ p = 0)))$›
**unfolding** *kp-periodic-def* **by** *simp+*
**have** ‹$\forall n.\ (\$c)\ n = (case\ n\ of\ 0 \Rightarrow False \mid Suc\ n' \Rightarrow c\ n')$›
**unfolding** *delay-def* **by** *simp*
**with** *2* **have**
*3*:‹$\forall n.\ (\$c)\ n = (case\ n\ of\ 0 \Rightarrow False \mid Suc\ n' \Rightarrow ((n' \geq k) \land ((n' - k)\ mod\ p = 0)))$›
**by** *presburger*
**have** ‹$\forall n.\ (\$c)\ n = ((n \geq k+1) \land ((n - (k+1))\ mod\ p = 0))$›
**proof** −
{ **fix** $n$
**have** ‹$(\$c)\ n = ((n \geq k+1) \land ((n - (k+1))\ mod\ p = 0))$›
**proof** (*cases* $n$)
**case** *0*
**thus** *?thesis* **by** (*simp add*: *3*)
**next**
**case** (*Suc* $n'$)
**with** *3* **have** ‹$(\$c)\ n = ((n' \geq k) \land ((n' - k)\ mod\ p = 0))$› **by** *simp*
**also have** ‹$\ldots = ((n-1 \geq k) \land ((n-1 - k)\ mod\ p = 0))$› **using** *Suc* **by** *auto*
**finally show** *?thesis* **using** *Suc* **by** *fastforce*
**qed**
} **thus** *?thesis* **..**
**qed**
**thus** *?thesis* **unfolding** *kp-periodic-def* **using** *1* **by** *simp*
**qed**

Get the number of ticks on a clock from the beginning up to instant n.

**definition** *ticks-up-to* :: ‹[*clock*, *nat*] ⇒ *nat*›
  **where** ‹*ticks-up-to c n* = *card* {*t*. *t* ≤ *n* ∧ *c t*}›

There cannot be more than n event occurrences during n instants.

**lemma** ‹*ticks-up-to c n* ≤ *Suc n*›
**proof** −
  **have** *finite*: ‹*finite* {*t*::*nat*. *t* ≤ *n*}› **by** *simp*
  **have** *incl*: ‹{*t*::*nat*. *t* ≤ *n* ∧ *c t*} ⊆ {*t*::*nat*. *t* ≤ *n*}› **by** *blast*
  **have** ‹*card* {*t*::*nat*. *t* ≤ *n*} = *Suc n*› **by** *simp*
  **with** *card-mono*[*OF finite incl*] **show** *?thesis* **unfolding** *ticks-up-to-def* **by** *simp*
**qed**

Counting event occurrences.

**definition** ‹*count b n* ≡ *if b then Suc n else n*›

The count of event occurrences cannot grow by more than one at each instant.

**lemma** *count-inc*: ‹*count b n* ≤ *Suc n*›
  **using** *count-def* **by** *simp*

Alternative definition of the number of event occurrences using fold.

**definition** *ticks-up-to-fold* :: ‹[*clock*, *nat*] ⇒ *nat*›
  **where** ‹*ticks-up-to-fold c n* = *fold count* (*map c* [*0*..<*Suc n*]) *0*›

Alternative definition of the number of event occurrences as a function.

**fun** *ticks-up-to-fun* :: ‹[*clock*, *nat*] ⇒ *nat*›
**where**
  ‹*ticks-up-to-fun c 0* = *count* (*c 0*) *0*›
| ‹*ticks-up-to-fun c* (*Suc n*) = *count* (*c* (*Suc n*)) (*ticks-up-to-fun c n*)›

Proof that the original definition and the function definition are equivalent. Use this to generate code.

**lemma** *ticks-up-to-is-fun*[*code*]: ‹*ticks-up-to c n* = *ticks-up-to-fun c n*›
**proof** (*induction n*)
  **case** *0*
    **have** ‹*ticks-up-to c 0* = *card* {*t*. *t* ≤ *0* ∧ *c t*}›
      **by** (*simp add:ticks-up-to-def*)
    **also have** ‹... = *card* {*t*. *t*=*0* ∧ *c t*}› **by** *simp*
    **also have** ‹... = (*if c 0 then 1 else 0*)›
      **by** (*simp add*: *Collect-conv-if*)
    **also have** ‹... = *ticks-up-to-fun c 0*›
      **using** *ticks-up-to-fun.simps*(*1*) *count-def* **by** *simp*
    **finally show** *?case* **.**
**next**
  **case** (*Suc n*)
    **show** *?case*

**proof** (*cases* ‹*c* (*Suc n*)›)
  **case** *True*
    **hence** ‹$\{t.\ t \leq Suc\ n \wedge c\ t\} = insert\ (Suc\ n)\ \{t.\ t \leq n \wedge c\ t\}$› **by** *auto*
    **hence** ‹*ticks-up-to c* (*Suc n*) = *Suc* (*ticks-up-to c n*)›
      **by** (*simp add*: *ticks-up-to-def*)
    **also have** ‹... = *Suc* (*ticks-up-to-fun c n*)› **using** *Suc.IH* **by** *simp*
    **finally show** *?thesis* **by** (*simp add*: *count-def* ‹*c* (*Suc n*)›)
**next**
  **case** *False*
    **hence** ‹$\{t.\ t \leq Suc\ n \wedge c\ t\} = \{t.\ t \leq n \wedge c\ t\}$› **using** *le-Suc-eq* **by** *blast*
    **hence** ‹*ticks-up-to c* (*Suc n*) = *ticks-up-to c n*›
      **by** (*simp add*: *ticks-up-to-def*)
    **also have** ‹... = *ticks-up-to-fun c n*› **using** *Suc.IH* **by** *simp*
    **finally show** *?thesis* **by** (*simp add*: *count-def* ‹¬*c* (*Suc n*)›)
**qed**
**qed**

Proof that the original definition and the definition using fold are equivalent.

**lemma** *ticks-up-to-is-fold*: ‹*ticks-up-to c n* = *ticks-up-to-fold c n*›
**proof** (*induction n*)
  **case** *0*
  **have** ‹*ticks-up-to c 0* = *card* $\{t.\ t \leq 0 \wedge c\ t\}$›
    **by** (*simp add*:*ticks-up-to-def*)
  **also have** ‹... = *card* $\{t.\ t=0 \wedge c\ t\}$› **by** *simp*
  **also have** ‹... = (*if c 0 then 1 else 0*)›
    **by** (*simp add*: *Collect-conv-if*)
  **also have** ‹... = *ticks-up-to-fold c 0*›
    **using** *ticks-up-to-fold-def count-def* **by** *simp*
  **finally show** *?case* **.**
**next**
  **case** (*Suc n*)
  **show** *?case*
  **proof** (*cases* ‹*c* (*Suc n*)›)
    **case** *True*
      **hence** ‹$\{t.\ t \leq Suc\ n \wedge c\ t\} = insert\ (Suc\ n)\ \{t.\ t \leq n \wedge c\ t\}$› **by** *auto*
      **hence** ‹*ticks-up-to c* (*Suc n*) = *Suc* (*ticks-up-to c n*)›
        **by** (*simp add*: *ticks-up-to-def*)
      **also have** ‹... = *Suc* (*ticks-up-to-fold c n*)› **using** *Suc.IH* **by** *simp*
      **finally show** *?thesis* **by** (*simp add*: *ticks-up-to-fold-def count-def* ‹*c* (*Suc n*)›)
  **next**
    **case** *False*
      **hence** ‹$\{t.\ t \leq Suc\ n \wedge c\ t\} = \{t.\ t \leq n \wedge c\ t\}$› **using** *le-Suc-eq* **by** *blast*
      **hence** ‹*ticks-up-to c* (*Suc n*) = *ticks-up-to c n*›
        **by** (*simp add*: *ticks-up-to-def*)
      **also have** ‹... = *ticks-up-to-fold c n*› **using** *Suc.IH* **by** *simp*
      **finally show** *?thesis* **by** (*simp add*: *ticks-up-to-fold-def count-def* ‹¬*c* (*Suc n*)›)
  **qed**

**qed**

Number of ticks during an n instant window starting at $k_0$.

**definition** *tick-count* ::‹[*clock*, *nat*, *nat*] ⇒ *nat*›
  **where** ‹$tick\text{-}count\ c\ k_0\ n \equiv card\ \{k.\ k_0 \leq k \wedge k < k_0+n \wedge c\ k\}$›

The number of ticks is monotonous with regard to the window width.

**lemma** *tick-count-mono*:
  **assumes** ‹$n' \geq n$›
    **shows** ‹$tick\text{-}count\ c\ t_0\ n' \geq tick\text{-}count\ c\ t_0\ n$›
**proof** −
  **have** *finite*: ‹$finite\ \{t{::}nat.\ t_0 \leq t \wedge t < t_0+n' \wedge c\ t\}$› **by** *simp*
  **from** *assms* **have** *incl*:
    ‹$\{t{::}nat.\ t_0 \leq t \wedge t < t_0+n \wedge c\ t\} \subseteq \{t{::}nat.\ t_0 \leq t \wedge t < t_0+n' \wedge c\ t\}$› **by** *auto*
  **have** ‹$card\ \{t{::}nat.\ t_0 \leq t \wedge t < t_0+n \wedge c\ t\}$
      $\leq card\ \{t{::}nat.\ t_0 \leq t \wedge t < t_0+n' \wedge c\ t\}$›
    **using** *card-mono*[*OF finite incl*] **.**
  **thus** *?thesis* **using** *tick-count-def* **by** *simp*
**qed**

The interval [t, t+n[ contains n instants.

**lemma** *card-interval*:‹$card\ \{t.\ t_0 \leq t \wedge t < t_0+n\} = n$›
**proof** (*induction n*)
  **case** *0*
  **then show** *?case* **by** *simp*
**next**
  **case** (*Suc n*)
  **have** ‹$\{t.\ t_0 \leq t \wedge t < t_0+(Suc\ n)\} = insert\ (t_0+n)\ \{t.\ t_0 \leq t \wedge t < t_0+n\}$› **by** *auto*
  **hence** ‹$card\ \{t.\ t_0 \leq t \wedge t < t_0+(Suc\ n)\} = Suc\ (card\ \{t.\ t_0 \leq t \wedge t < t_0+n\})$› **by** *simp*
  **with** *Suc.IH* **show** *?case* **by** *simp*
**qed**

There cannot be more than n occurrences of an event in an interval of n instants.

**lemma** *tick-count-bound*: ‹$tick\text{-}count\ c\ t_0\ n \leq n$›
**proof** −
  **have** *finite*: ‹$finite\ \{t.\ t_0 \leq t \wedge t < t_0+n\}$› **by** *simp*
  **have** *incl*: ‹$\{t.\ t_0 \leq t \wedge t < t_0+n \wedge c\ t\} \subseteq \{t.\ t_0 \leq t \wedge t < t_0+n\}$› **by** *blast*
  **show** *?thesis* **using** *tick-count-def card-interval card-mono*[*OF finite incl*] **by** *simp*
**qed**

No event occurrence occur in 0 instant.

**lemma** *tick-count-0*[*code*]: ‹$tick\text{-}count\ c\ t_0\ 0 = 0$›
  **unfolding** *tick-count-def* **by** *simp*

Event occurrences starting from instant 0 are event occurrences from the beginning.

**lemma** *tick-count-orig*[*code*]:
‹$tick\text{-}count\ c\ 0\ (Suc\ n) = ticks\text{-}up\text{-}to\ c\ n$›
**unfolding** *tick-count-def ticks-up-to-def*
**using** *less-Suc-eq-le* **by** *simp*

Counting event occurrences between two instants is simply subtracting occurrence counts from the beginning.

**lemma** *tick-count-diff*[*code*]:
‹$tick\text{-}count\ c\ (Suc\ t_0)\ n = (ticks\text{-}up\text{-}to\ c\ (t_0{+}n)) - (ticks\text{-}up\text{-}to\ c\ t_0)$›
**proof** −
**have** *incl*: ‹$\{t.\ t \le t_0 \wedge c\ t\} \subseteq \{t.\ t \le t_0{+}n \wedge c\ t\}$› **by** *auto*
**have** ‹$\{t.\ (Suc\ t_0) \le t \wedge t < (Suc\ t_0){+}n \wedge c\ t\}$
$= \{t.\ t \le t_0{+}n \wedge c\ t\} - \{t.\ t \le t_0 \wedge c\ t\}$› **by** *auto*
**hence** ‹$card\ \{t.\ (Suc\ t_0) \le t \wedge t < (Suc\ t_0){+}n \wedge c\ t\}$
$= card\ \{t.\ t \le t_0{+}n \wedge c\ t\} - card\ \{t.\ t \le t_0 \wedge c\ t\}$›
**by** (*simp add*: *card-Diff-subset incl*)
**thus** *?thesis* **unfolding** *tick-count-def ticks-up-to-def* **.**
**qed**

The merge of two clocks has less ticks than the union of the ticks of the two clocks.

**lemma** *tick-count-merge*:
‹$tick\text{-}count\ (c{\oplus}c')\ t_0\ n \ \le tick\text{-}count\ c\ t_0\ n + tick\text{-}count\ c'\ t_0\ n$›
**proof** −
**have** ‹$\{t{::}nat.\ t_0 \le t \wedge t < t_0{+}n \wedge ((c{\oplus}c')\ t)\}$
$= \{t{::}nat.\ t_0 \le t \wedge t < t_0{+}n \wedge c\ t\} \cup \{t{::}nat.\ t_0 \le t \wedge t < t_0{+}n \wedge c'\ t\}$›
**using** *merge-def* **by** *auto*
**hence** ‹$card\ \{t{::}nat.\ t_0 \le t \wedge t < t_0{+}n \wedge ((c{\oplus}c')\ t)\}$
$\le card\ \{t{::}nat.\ t_0 \le t \wedge t < t_0{+}n \wedge c\ t\}$
$+ card\ \{t{::}nat.\ t_0 \le t \wedge t < t_0{+}n \wedge c'\ t\}$› **by** (*simp add*: *card-Un-le*)
**thus** *?thesis* **unfolding** *tick-count-def* **.**
**qed**

# 4 Bounded clocks

An (n,m)-bounded clock does not tick more than m times in a n interval of width n.

**definition** *bounded* :: ‹[*nat*, *nat*, *clock*] ⇒ *bool*›
**where** ‹$bounded\ n\ m\ c \equiv \forall t.\ tick\text{-}count\ c\ t\ n \le m$›

All clocks are (n,n)-bounded.

**lemma** *bounded-n*: ‹$bounded\ n\ n\ c$›
**unfolding** *bounded-def* **using** *tick-count-bound* **by** (*simp add*: *le-imp-less-Suc*)

A sporadic clock is bounded.

```
lemma spor-bound:
  assumes ‹∀ t::nat. c t ⟶ (∀ t'. (t < t' ∧ t' ≤ t+n) ⟶ ¬(c t'))›
  shows ‹∀ t::nat. card {t'. t ≤ t' ∧ t' ≤ t+n ∧ c t'} ≤ 1›
proof –
  { fix t::nat
    have ‹card {t'. t ≤ t' ∧ t' ≤ t+n ∧ c t'} ≤ 1›
    proof (cases ‹c t›)
      case True
        with assms have ‹∀ t'. (t < t' ∧ t' ≤ t+n) ⟶ ¬(c t')› by simp
        hence empty: ‹card {t'. t < t' ∧ t' ≤ t+n ∧ c t'} = 0› by simp
        have finite: ‹finite {t'. t < t' ∧ t' ≤ t+n ∧ c t'}› by simp
        have notin: ‹t ∉ {t'. t < t' ∧ t' ≤ t+n ∧ c t'}› by simp
        have ‹{t'. t ≤ t' ∧ t' ≤ t+n ∧ c t'}
            = insert t {t'. t < t' ∧ t' ≤ t+n ∧ c t'}› using ‹c t› by auto
        hence ‹card {t'. t ≤ t' ∧ t' ≤ t+n ∧ c t'} = 1›
          using empty card-insert-disjoint[OF finite notin] by simp
        then show ?thesis by simp
    next
      case False
      then show ?thesis
      proof(cases ‹∃ tt. t < tt ∧ tt ≤ t+n ∧ c tt›)
        case True

        hence ‹∃ ttmin. t < ttmin ∧ ttmin ≤ t+n ∧ c ttmin
              ∧ (∀ tt'. (t < tt' ∧ tt' ≤ t+n ∧ c tt') ⟶ ttmin ≤ tt')›
          by (metis add-lessD1 add-less-mono1 assms le-eq-less-or-eq
              le-refl less-imp-le-nat nat-le-iff-add nat-le-linear)
        from this obtain ttmin where
          tmin: ‹t < ttmin ∧ ttmin ≤ t+n ∧ c ttmin
                ∧ (∀ tt'. (t < tt' ∧ tt' ≤ t+n ∧ c tt') ⟶ ttmin ≤ tt')› by blast
        hence tick:‹c ttmin› by simp
         with assms have notick:‹(∀ t'. ttmin < t' ∧ t' ≤ ttmin + n ⟶ ¬ c t')›
by simp
        have ‹∀ t'. (t < t' ∧ t' < ttmin) ⟶ ¬c t'› using tmin ‹¬c t› by auto
        moreover from notick tmin have
          ‹∀ t'. (ttmin < t' ∧ t' ≤ t+n) ⟶ ¬c t'› by auto
        ultimately have ‹∀ t'::nat. (t ≤ t' ∧ t' ≤ t+n ∧ c t') ⟶ t' = ttmin›
          using tick tmin ‹¬c t› le-eq-less-or-eq by auto
        hence ‹{t'. t ≤ t' ∧ t' ≤ t+n ∧ c t'} = {ttmin}› using tmin by fastforce
        hence ‹card {t'. t ≤ t' ∧ t' ≤ t+n ∧ c t'} = 1› by simp
        thus ?thesis by simp
      next
        case False
          with ‹¬c t› have ‹∀ t'. t ≤ t' ∧ t' ≤ t+n ⟶ ¬c t'›
            using nat-less-le by blast
          hence ‹card {t'. t ≤ t' ∧ t' ≤ t+n ∧ c t'} = 0› by simp
          thus ?thesis by linarith
      qed
    qed
```

```
  } thus ?thesis ..
qed
```

A sporadic clock is bounded.

```
lemma spor-bound':
  assumes ‹∀ t::nat. c t ⟶ (∀ t'. (t < t' ∧ c t') ⟶ t' > t+n)›
  shows ‹∀ t::nat. card {t'. t ≤ t' ∧ t' ≤ t+n ∧ c t'} ≤ 1›
proof −
  { fix t::nat
    have ‹card {t'. t ≤ t' ∧ t' ≤ t+n ∧ c t'} ≤ 1›
    proof (cases ‹c t›)
      case True
        with assms have ‹∀ t'. (t < t' ∧ c t') ⟶ t' > t+n› by simp
        hence empty: ‹card {t'. t < t' ∧ t' ≤ t+n ∧ c t'} = 0› by auto
        have finite: ‹finite {t'. t < t' ∧ t' ≤ t+n ∧ c t'}› by simp
        have notin: ‹t ∉ {t'. t < t' ∧ t' ≤ t+n ∧ c t'}› by simp
        have ‹{t'. t ≤ t' ∧ t' ≤ t+n ∧ c t'}
            = insert t {t'. t < t' ∧ t' ≤ t+n ∧ c t'}› using ‹c t› by auto
        hence ‹card {t'. t ≤ t' ∧ t' ≤ t+n ∧ c t'} = 1›
          using empty card-insert-disjoint[OF finite notin] by simp
        then show ?thesis by simp
    next
      case False
      then show ?thesis
      proof(cases ‹∃ tt. t < tt ∧ tt ≤ t+n ∧ c tt›)
        case True

        hence ‹∃ ttmin. t < ttmin ∧ ttmin ≤ t+n ∧ c ttmin
              ∧ (∀ tt'. (t < tt' ∧ tt' ≤ t+n ∧ c tt') ⟶ ttmin ≤ tt')›
         by (metis add-lessD1 add-less-mono1 assms le-Suc-ex le-eq-less-or-eq le-refl
less-imp-le-nat nat-le-linear nat-neq-iff)
        from this obtain ttmin where
          tmin: ‹t < ttmin ∧ ttmin ≤ t+n ∧ c ttmin
                ∧ (∀ tt'. (t < tt' ∧ tt' ≤ t+n ∧ c tt') ⟶ ttmin ≤ tt')› by blast
        hence tick:‹c ttmin› by simp
        with assms have notick:‹(∀ t'. ttmin < t' ∧ c t' ⟶ t' > ttmin + n)› by
simp
        have ‹∀ t'. (t < t' ∧ t' < ttmin) ⟶ ¬c t'› using tmin ‹¬c t› by auto
        moreover from notick tmin have
          ‹∀ t'. (ttmin < t' ∧ t' ≤ t+n) ⟶ ¬c t'› by auto
        ultimately have ‹∀ t'::nat. (t ≤ t' ∧ t' ≤ t+n ∧ c t') ⟶ t' = ttmin›
          using tick tmin ‹¬c t› le-eq-less-or-eq by auto
        hence ‹{t'. t ≤ t' ∧ t' ≤ t+n ∧ c t'} = {ttmin}› using tmin by fastforce
        hence ‹card {t'. t ≤ t' ∧ t' ≤ t+n ∧ c t'} = 1› by simp
        thus ?thesis by simp
      next
        case False
          with ‹¬c t› have ‹∀ t'. t ≤ t' ∧ t' ≤ t+n ⟶ ¬c t'›
            using nat-less-le by blast
```

```
        hence ‹card {t'. t ≤ t' ∧ t' ≤ t+n ∧ c t'} = 0› by simp
        thus ?thesis by linarith
      qed
    qed
  } thus ?thesis ..
qed
```

An n-sporadic clock is (n+1, 1)-bounded.

```
lemma spor-bounded:
  assumes ‹p-sporadic n c›
    shows ‹bounded (n+1) 1 c›
proof -
  from assms have ‹∀t. c t ⟶ (∀t'. (t < t' ∧ c t') ⟶ t' > t+n)›
    using p-sporadic-def by simp
  from spor-bound'[OF this] have ‹∀t. card {t'. t ≤ t' ∧ t' ≤ t+n ∧ c t'} ≤ 1› .
  hence ‹∀t. card {t'. t ≤ t' ∧ t' < Suc (t+n) ∧ c t'} ≤ 1›
    using less-Suc-eq-le by auto
  hence ‹∀t. card {t'. t ≤ t' ∧ t' < t + Suc n ∧ c t'} ≤ 1› by auto
  thus ?thesis unfolding bounded-def tick-count-def Suc-eq-plus1 .
qed
```

An n-sporadic clock is (n+2, 2)-bounded.

```
lemma spor-bounded2:
  assumes ‹p-sporadic n c›
    shows ‹bounded (n+2) 2 c›
proof -
  from spor-bounded[OF assms] have
      *:‹∀t. card {t'. t ≤ t' ∧ t' < t + Suc n ∧ c t'} ≤ 1›
    unfolding bounded-def tick-count-def by simp
  hence ‹∀t. card {t'. t ≤ t' ∧ t' < Suc (t + Suc n) ∧ c t'} ≤ Suc 1›
  proof -
    { fix t::nat
      from * have **:‹card {t'. t ≤ t' ∧ t' < t + Suc n ∧ c t'} ≤ 1› by simp
      have ‹card {t'. t ≤ t' ∧ t' < Suc (t + Suc n) ∧ c t'} ≤ Suc 1›
      proof (cases ‹c (t + Suc n)›)
        case True
          hence ‹{t'. t ≤ t' ∧ t' < Suc (t + Suc n) ∧ c t'}
                = insert (t+Suc n) {t'. t ≤ t' ∧ t' < t + Suc n ∧ c t'}› by auto
          hence ‹card {t'. t ≤ t' ∧ t' < Suc (t + Suc n) ∧ c t'}
                = Suc (card {t'. t ≤ t' ∧ t' < t + Suc n ∧ c t'})› by simp
          thus ?thesis using ** by simp
      next
        case False
          hence ‹{t'. t ≤ t' ∧ t' < Suc (t + Suc n) ∧ c t'}
                = {t'. t ≤ t' ∧ t' < t + Suc n ∧ c t'}› using less-Suc-eq by blast
          hence ‹card {t'. t ≤ t' ∧ t' < Suc (t + Suc n) ∧ c t'}
                 = (card {t'. t ≤ t' ∧ t' < t + Suc n ∧ c t'})› by simp
          thus ?thesis using ** by simp
      qed
```

} **thus** *?thesis* ..
**qed**
**thus** *?thesis* **unfolding** *bounded-def tick-count-def*
**by** (*metis Suc-1 add-Suc-right Suc-eq-plus1*)
**qed**

A bounded clock on an interval is also bounded on a narrower interval.

**lemma** *bounded-less*:
**assumes** ‹*bounded* $n'$ *m c*›
**and** ‹$n' \geq n$›
**shows** ‹*bounded n m c*›
**using** *assms*(*1*) **unfolding** *bounded-def*
**using** *tick-count-mono*[*OF assms*(*2*)] *order-trans* **by** *blast*

The merge of two bounded clocks is bounded.

**lemma** *bounded-merge*:
**assumes** ‹*bounded n m c*›
**and** ‹*bounded* $n'$ $m'$ $c'$›
**and** ‹$n' \geq n$›
**shows** ‹*bounded n* ($m$+$m'$) ($c \oplus c'$)›
**using** *tick-count-merge bounded-less*[*OF assms*(*2*,*3*)] *assms*(*1*,*2*) *add-mono order-trans*
**unfolding** *bounded-def* **by** *blast*

The merge of two sporadic clocks is bounded.

**lemma** *sporadic-bounded1*:
**assumes** ‹*p-sporadic n c*›
**and** ‹*p-sporadic* $n'$ $c'$›
**and** ‹$n' \geq n$›
**shows** ‹*bounded* ($n$+*1*) *2* ($c \oplus c'$)›
**proof** −
**have** *1*:‹*bounded* ($n$+*1*) *1 c*› **using** *spor-bounded*[*OF assms*(*1*)] .
**have** *2*:‹*bounded* ($n'$+*1*) *1* $c'$› **using** *spor-bounded*[*OF assms*(*2*)] .
**from** *assms*(*3*) **have** *3*:‹$n'$+*1* $\geq$ $n$+*1*› **by** *simp*
**have** ‹*1*+*1* = (*2*::*nat*)› **by** *simp*
**with** *bounded-merge*[*OF 1 2 3*] **show** *?thesis* **by** *metis*
**qed**

## 4.1 Main theorem

The merge of two sporadic clocks is bounded on the min of the bounding intervals.

**theorem** *sporadic-bounded-min*:
**assumes** ‹*p-sporadic n c*›
**and** ‹*p-sporadic* $n'$ $c'$›
**shows** ‹*bounded* ((*min n* $n'$)+*1*) *2* ($c \oplus c'$)›
**proof** (*cases* ‹$n \leq n'$›)
**case** *True*
**hence** ‹*min n* $n'$ = $n$› **by** *simp*

**thus** *?thesis* **using** *sporadic-bounded1*[*OF assms True*] **by** *simp*
**next**
**case** *False*
**hence** *1*:‹$n' = min\ n\ n'$› **and** *2*:‹$n' \leq n$› **by** *simp+*
**from** *sporadic-bounded1*[*OF assms*(*2*) *assms*(*1*) *2*] *1* **show** *?thesis* **using** *merge-comm* **by** *simp*
**qed**

# 5 Logical time

Logical time is a natural number that is attached to instants. Logical time can stay constant for an arbitrary number of instants, but it cannot decrease. When logical time stays constant for an infinite number of instants, we have a Zeno condition.

**typedef** *time* = ‹{*t*::*nat*⇒*nat*. *mono t*}›
**using** *mono-Suc* **by** *blast*

**setup-lifting** *type-definition-time*

A chronometric clock is a clock associated with a time line.

**type-synonym** *chronoclock* = ‹*clock*×*time*›

@term$c\ \nabla\ t$ tells whether chronometric clock $c$ ticks at instant $t$.

**definition** *ticks* :: ‹[*chronoclock*, *nat*] ⇒ *bool*› (**infix** ‹∇› *60*)
**where** ‹$c\ \nabla\ t \equiv (fst\ c)\ t$›

@term$c_t$ is the logical time on clock $c$ at instant $t$.

**lift-definition** *time-at* :: ‹[*chronoclock*, *nat*] ⇒ *nat*› (‹-_› [*60*, *60*])
**is** ‹$\lambda c\ t.\ (snd\ c)\ t$› **.**

**lemmas** *chronoclocks-simp*[*simp*] = *ticks-def time-at-def*

As consequence of the definition of the *time* type, (∇) is monotonous for any clock.

**lemma** *mono-chronotime*:
‹*mono* (*time-at c*)› **using** *Rep-time* **by** *auto*

An event occurs at a given time if the clock ticks at some instant at that time.

**definition** *occurs* :: ‹[*nat*, *chronoclock*] ⇒ *bool*›
**where** ‹$occurs\ n\ c \equiv \exists\, k.\ (c\ \nabla\ k \wedge c_k = n)$›

An event occurs once at a given time if the clock ticks at exactly one instant at that time.

**definition** *occurs-once* :: ‹[*nat*, *chronoclock*] ⇒ *bool*›
**where** ‹$occurs\text{-}once\ n\ c \equiv \exists !k.\ (c\ \nabla\ k \wedge c_k = n)$›

**lemma** *occurs-once-occurs*:
  ‹$occurs\text{-}once\ n\ c \Longrightarrow occurs\ n\ c$›
**unfolding** *occurs-once-def occurs-def* **by** *blast*

A clock is strict at a given time if it ticks at most once at that time.

**definition** *strict-at* :: ‹$[nat,\ chronoclock] \Rightarrow bool$›
  **where** ‹$strict\text{-}at\ n\ c \equiv (occurs\ n\ c \longrightarrow occurs\text{-}once\ n\ c)$›

**definition** *strict-clock* :: ‹$chronoclock \Rightarrow bool$›
  **where** ‹$strict\text{-}clock\ c \equiv (\forall n.\ strict\text{-}at\ n\ c)$›

## 5.1 Chrono-periodic and chrono-sporadic clocks

The introduction of logical time allows us to define periodicity and sporadicity on logical time instead of instant index.

**definition** *kp-chronoperiodic* :: ‹$[nat,\ nat,\ chronoclock] \Rightarrow bool$›
  **where** ‹$kp\text{-}chronoperiodic\ k\ p\ c \equiv (p > 0) \wedge (\forall n.\ occurs\ n\ c = ((n \geq k) \wedge ((n - k)\ mod\ p = 0)))$›

**definition** *p-chronoperiodic* :: ‹$[nat,\ chronoclock] \Rightarrow bool$›
  **where** ‹$p\text{-}chronoperiodic\ p\ c \equiv \exists k.\ kp\text{-}chronoperiodic\ k\ p\ c$›

**definition** *chronoperiodic* :: ‹$[chronoclock] \Rightarrow bool$›
  **where** ‹$chronoperiodic\ c \equiv \exists p.\ p\text{-}chronoperiodic\ p\ c$›

A clock is strictly chronoperiodic if it ticks only once at the logical times when it ticks.

**definition** *chronoperiodic-strict* :: ‹$[chronoclock] \Rightarrow bool$›
  **where** ‹$chronoperiodic\text{-}strict\ c \equiv chronoperiodic\ c \wedge strict\text{-}clock\ c$›

**definition** *p-chronoperiodic-strict* :: ‹$[nat,\ chronoclock] \Rightarrow bool$›
  **where** ‹$p\text{-}chronoperiodic\text{-}strict\ p\ c \equiv p\text{-}chronoperiodic\ p\ c \wedge strict\text{-}clock\ c$›

**lemma** ‹$chronoperiodic\text{-}strict\ c \Longrightarrow chronoperiodic\ c$›
  **unfolding** *chronoperiodic-strict-def* **by** *simp*

**definition** *p-chronosporadic* :: ‹$[nat,\ chronoclock] \Rightarrow bool$›
  **where** ‹$p\text{-}chronosporadic\ p\ c \equiv$
    $\forall t.\ occurs\ t\ c \longrightarrow (\forall t'.\ (t' > t \wedge occurs\ t'\ c) \longrightarrow t' > t + p)$›

**definition** ‹$p\text{-}chronosporadic\text{-}strict\ p\ c \equiv p\text{-}chronosporadic\ p\ c \wedge strict\text{-}clock\ c$›

**definition** ‹$chronosporadic\ c \equiv (\exists p > 0.\ p\text{-}chronosporadic\ p\ c)$›

**definition** ‹$chronosporadic\text{-}strict\ c \equiv chronosporadic\ c \wedge strict\text{-}clock\ c$›

**lemma** *chrono-periodic-suc-sporadic*:

**assumes** ‹*p-chronoperiodic* $(p+1)$ *c*›
  **shows** ‹*p-chronosporadic p c*›
**proof** –
  **from** *assms p-chronoperiodic-def* **obtain** *k*
    **where** ‹*kp-chronoperiodic k* $(p+1)$ *c*› **by** *blast*
  **hence** ∗:‹$\forall n.$ *occurs n c* $= ((n \geq k) \wedge ((n - k)$ *mod* $(p+1) = 0))$›
    **unfolding** *kp-chronoperiodic-def* **by** *simp*
  **with** *mod-offset-sporadic′*[*of k* - ‹$p+1$›] **have**
    ‹$\forall n.$ *occurs n c* $\longrightarrow (\forall n'. (n < n' \wedge ((n'-k)$ *mod* $(p+1) = 0)) \longrightarrow n' \geq n+p+1)$›
  **by** *simp*
  **thus** *?thesis* **unfolding** *p-chronosporadic-def* **by** (*simp add*: ∗ *Suc-le-lessD*)
**qed**

**lemma** *chrono-periodic-suc-sporadic-strict*:
  **assumes** ‹*p-chronoperiodic-strict* $(p+1)$ *c*›
    **shows** ‹*p-chronosporadic-strict p c*›
  **using** *assms chrono-periodic-suc-sporadic*
      *p-chronoperiodic-strict-def p-chronosporadic-strict-def*
  **by** *simp*

Number of ticks up to a given logical time. This counts distinct ticks that happen at the same logical time.

**definition** *chrono-dense-up-to* ::‹[*chronoclock*, *nat*] ⇒ *nat*›
  **where** ‹*chrono-dense-up-to c n* = *card* $\{t.\ c_t \leq n \wedge c \nabla t\}$›

A clock is Zeno if it ticks an infinite number of times in a finite amount of time.

**definition** *zeno-clock* :: ‹*chronoclock* ⇒ *bool*›
  **where** ‹*zeno-clock c* ≡ $(\exists \omega.$ *infinite* $\{t.\ c_t \leq \omega \wedge c \nabla t\})$›

Number of occurrences of an event up to a given logical time. This does not count separately ticks that occur at the same logical time.

**definition** *chrono-up-to* ::‹[*chronoclock*, *nat*] ⇒ *nat*›
  **where** ‹*chrono-up-to c n* = *card* $\{t.\ t \leq n \wedge$ *occurs t c*$\}$›

For any time n, a non Zeno clock has less occurrences than ticks up to n. This is also true for Zeno clock, but we count ticks and occurrences using *card*, and in Isabelle/HOL, the cardinal of an infinite set is 0, so the inequality breaks when there are infinitely many ticks before a given time.

**lemma** *not-zeno-sparse*:
  **assumes** ‹¬*zeno-clock c*›
    **shows** ‹*chrono-up-to c n* ≤ *chrono-dense-up-to c n*›
**proof** –
  **from** *assms* **have** ‹*finite* $\{t.\ c_t \leq n \wedge c \nabla t\}$›
    **unfolding** *zeno-clock-def* **by** *simp*
  **moreover from** *occurs-def* **have**
    ‹$\exists f. \forall t.\ t \leq n \wedge$ *occurs t c* $\longrightarrow$

$(\exists k.\ f\,k = t \wedge c_k \leq n \wedge c \nabla k)$› **by** *auto*
**hence**
‹$\exists f.\ \forall t \in \{t.\ t \leq n \wedge occurs\ t\ c\}.$
$\exists k.\ f\,k = t \wedge k \in \{k.\ c_k \leq n \wedge c \nabla k\}$› **by** *simp*
**hence** ‹$\exists f.\ \{t.\ t \leq n \wedge occurs\ t\ c\} \subseteq image\ f\ \{k.\ c_k \leq n \wedge c \nabla k\}$›
**by** *fastforce*
**ultimately have** ‹$card\ \{t.\ t \leq n \wedge occurs\ t\ c\} \leq card\ \{k.\ c_k \leq n \wedge c \nabla k\}$›
**using** *surj-card-le* **by** *blast*
**thus** *?thesis*
**unfolding** *chrono-up-to-def chrono-dense-up-to-def occurs-def* **by** *simp*
**qed**

Number of event occurrences during a time window.

**definition** *occurrence-count* :: ‹$[chronoclock, nat, nat] \Rightarrow nat$›
**where** ‹$occurrence\text{-}count\ c\ t_0\ d \equiv card\ \{t.\ t_0 \leq t \wedge t < t_0 + d \wedge occurs\ t\ c\}$›

The number of event occurrences is monotonous with regard to the window width.

**lemma** *occ-count-mono*:
**assumes** ‹$d' \geq d$›
**shows** ‹$occurrence\text{-}count\ c\ t_0\ d' \geq occurrence\text{-}count\ c\ t_0\ d$›
**proof** −
**have** *finite*: ‹$finite\ \{t::nat.\ t_0 \leq t \wedge t < t_0+d' \wedge occurs\ t\ c\}$› **by** *simp*
**from** *assms* **have** *incl*:
‹$\{t::nat.\ t_0 \leq t \wedge t < t_0+d \wedge occurs\ t\ c\} \subseteq \{t::nat.\ t_0 \leq t \wedge t < t_0+d' \wedge occurs\ t\ c\}$› **by** *auto*
**have** ‹$card\ \{t::nat.\ t_0 \leq t \wedge t < t_0+d \wedge occurs\ t\ c\}$
$\leq card\ \{t::nat.\ t_0 \leq t \wedge t < t_0+d' \wedge occurs\ t\ c\}$›
**using** *card-mono*[*OF finite incl*] **.**
**thus** *?thesis* **using** *occurrence-count-def* **by** *simp*
**qed**

# 6 Tests

**abbreviation** ‹$c1::clock \equiv (\lambda t.\ t \geq 1 \wedge (t-1)\ mod\ 2 = 0)$›
**abbreviation** ‹$c2::clock \equiv (\lambda t.\ t \geq 2 \wedge (t-2)\ mod\ 3 = 0)$›

**value** ‹*c1 0*›
**value** ‹*c1 1*›
**value** ‹*c1 2*›
**value** ‹*c1 3*›

**value** ‹*c2 0*›
**value** ‹*c2 1*›
**value** ‹*c2 2*›
**value** ‹*c2 3*›
**value** ‹*c2 4*›
**value** ‹*c2 5*›

```
lemma ‹kp-periodic 1 2 c1›
  using kp-periodic-def by simp

lemma ‹kp-periodic 2 3 c2›
  using kp-periodic-def by simp

abbreviation ‹c3 ≡ c1 ⊕ c2›

value ‹map c1 [0,1,2,3,4,5,6,7,8,9,10]›
value ‹map ($c1) [0,1,2,3,4,5,6,7,8,9,10,11]›
value ‹map c2 [0,1,2,3,4,5,6,7,8,9,10]›
value ‹map c3 [0,1,2,3,4,5,6,7,8,9,10]›

lemma interv-2:‹{t::nat. t0 ≤ t ∧ t < t0 + 2 ∧ 1 ≤ t ∧ (t − 1) mod 2 = 0} =
{t. (t = t0 ∨ t = t0 + 1) ∧ 1 ≤ t ∧ (t − 1) mod 2 = 0}›
  by auto

lemma ‹bounded 2 1 c1›
proof −
  have ‹∀ t. tick-count c1 t 2 ≤ 1›
  proof −
    { fix t0::nat
      have ‹tick-count c1 t0 2 ≤ 1›
      proof (cases t0)
        case 0
          hence ‹tick-count c1 t0 2 = ticks-up-to c1 1›
            using tick-count-orig by (simp add: numeral-2-eq-2)
          also have ‹... = card {t::nat. t ≤ 1 ∧ 1 ≤ t ∧ (t−1) mod 2 = 0}›
            unfolding ticks-up-to-def by simp
          also have ‹... ≤  card {t::nat. t ≤ 1 ∧ 1 ≤ t}›
            by (metis (mono-tags, lifting) Collect-cong
                cancel-comm-monoid-add-class.diff-cancel le-antisym le-refl mod-0)
          also have ‹... = card {t::nat. t = 1}› by (metis le-antisym order-refl)
          also have ‹... = 1› by simp
          finally show ?thesis .
      next
        case (Suc nat)
          then show ?thesis
          proof (cases ‹(t0−1) mod 2 = 0›)
            case True
              with Suc have ‹t0 mod 2 ≠ 0› by arith
               hence ‹{t. (t = t0 ∨ t = t0 + 1) ∧ 1 ≤ t ∧ (t − 1) mod 2 = 0} =
{t0}›
                using True  by auto
             hence ‹{t. t0 ≤ t ∧ t < t0 + 2 ∧ 1 ≤ t ∧ (t − 1) mod 2 = 0} = {t0}›
                using interv-2 by simp
              thus ?thesis unfolding tick-count-def by simp
          next
```

**case** *False*
**with** *Suc* **have** ‹$t_0$ *mod 2 = 0*› **by** *arith*
**hence** ‹$\{t.\ (t = t_0 \vee t = t_0 + 1) \wedge 1 \leq t \wedge (t - 1)\ mod\ 2 = 0\} = \{t_0{+}1\}$›
**by** *auto*
**hence** ‹$\{t.\ t_0 \leq t \wedge t < t_0 + 2 \wedge 1 \leq t \wedge (t - 1)\ mod\ 2 = 0\} = \{t_0{+}1\}$›
**using** *interv-2* **by** *simp*
**thus** *?thesis* **unfolding** *tick-count-def* **by** *simp*
**qed**
**qed**
**}**
**thus** *?thesis* **..**
**qed**
**thus** *?thesis* **using** *bounded-def* **by** *simp*
**qed**

**end**