Clocks

Frédéric Boulanger

February 22, 2020

Contents


```
theory Clocks

imports Main

begin
```

1 Basic definitions

Time is represented as the natural numbers. A clock represents an event that may occur or not at any time. 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) ∧ (∀n. c n = ((n ≥ k) ∧ ((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 ≥ 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 ≡ (∃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 ≡ (∃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 ≡ (∀t. c t ⟶ (∀t'. (t < t' ∧ t' ≤ t+p) ⟶ ¬(c t')))⟩
```

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_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
```

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

```
lemma periodic_suc_sporadic:
  assumes ‹p_periodic (Suc p) c›
    shows ‹p_sporadic p c›
proof -
  from assms p_periodic_def obtain k
    where ‹kp_periodic k (Suc p) c› by blast
  thus ?thesis
    using assms kp_periodic_def p_sporadic_def mod_offset_sporadic by auto
qed
```

3 Merging clocks

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

```
definition merge :: ‹[clock, clock] ⇒ clock› (infix ‹⊕› 60)
  where ‹c1 ⊕ c2 ≡ λt. c1 t ∨ c2 t›
```

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

```
lemma merge_no_sporadic:
  ‹∃c c'. sporadic c ∧ sporadic c' ∧ ¬sporadic (c⊕c')›
proof -
  define c :: clock where ‹c = (λt. t mod 2 = 0)›
  define c' :: clock where ‹c' = (λt. t ≥ 1 ∧ (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 ‹¬sporadic (c⊕c')›
  proof -
    { assume ‹sporadic (c ⊕ c')›
      from this obtain p where *:‹p > 0› and ‹p_sporadic p (c ⊕ c')›
        using sporadic_def by blast
      hence ‹∀t. (c⊕c') t ⟶ (∀t'. (t < t' ∧ t' ≤ t+p) ⟶ ¬((c⊕c') t'))›
        by (simp add:p_sporadic_def)
      moreover have ‹(c⊕c') 0› using c_def c'_def merge_def by simp
```

```
      moreover have ‹(c⊕c') 1› using c_def c'_def merge_def by simp
      ultimately have False by (simp add: "*" Suc_leI)
    } thus ?thesis ..
  qed
  with 1 and 2 show ?thesis by blast
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 ≤ Suc n ∧ c t} = insert (Suc n) {t. t ≤ n ∧ 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 ≤ Suc n ∧ c t} = {t. t ≤ n ∧ 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
```

Number of event occurrences during an n instant window starting at t_0.

```
definition tick_count ::‹[clock, nat, nat] ⇒ nat›
  where ‹tick_count c t0 n ≡ card {t. t0 ≤ t ∧ t < t0+n ∧ c t}›
```

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

```
lemma tick_count_mono:
  assumes ‹n' ≥ n›
    shows ‹tick_count c t0 n' ≥ tick_count c t0 n›
proof -
  have finite: ‹finite {t::nat. t0 ≤ t ∧ t < t0+n' ∧ c t}› by simp
  from assms have incl:
    ‹{t::nat. t0 ≤ t ∧ t < t0+n ∧ c t} ⊆ {t::nat. t0 ≤ t ∧ t < t0+n' ∧ c t}› by auto
  have ‹card {t::nat. t0 ≤ t ∧ t < t0+n ∧ c t}
        ≤ card {t::nat. t0 ≤ t ∧ t < t0+n' ∧ 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. t0 ≤ t ∧ t < t0+n} = n›
proof (induction n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  have ‹{t. t0 ≤ t ∧ t < t0+(Suc n)} = insert (t0+n) {t. t0 ≤ t ∧ t < t0+n}› by auto
  hence ‹card {t. t0 ≤ t ∧ t < t0+(Suc n)} = Suc (card {t. t0 ≤ t ∧ t < t0+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_count c t0 n ≤ n›
proof -
  have finite: ‹finite {t. t0 ≤ t ∧ t < t0+n}› by simp
  have incl: ‹{t. t0 ≤ t ∧ t < t0+n ∧ c t} ⊆ {t. t0 ≤ t ∧ t < t0+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_count c t0 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_count c 0 (Suc n) = ticks_up_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_count c (Suc t0) n = (ticks_up_to c (t0+n)) - (ticks_up_to c t0)⟩
proof -
  have incl: ⟨{t. t ≤ t0 ∧ c t} ⊆ {t. t ≤ t0+n ∧ c t}⟩ by auto
  have ⟨{t. (Suc t0) ≤ t ∧ t < (Suc t0)+n ∧ c t}
      = {t. t ≤ t0+n ∧ c t} - {t. t ≤ t0 ∧ c t}⟩ by auto
  hence ⟨card {t. (Suc t0) ≤ t ∧ t < (Suc t0)+n ∧ c t}
       = card {t. t ≤ t0+n ∧ c t} - card {t. t ≤ t0 ∧ 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_count (c⊕c') t0 n  ≤ tick_count c t0 n + tick_count c' t0 n⟩
proof -
  have ⟨{t::nat. t0 ≤ t ∧ t < t0+n ∧ ((c⊕c') t)}
        = {t::nat. t0 ≤ t ∧ t < t0+n ∧ c t} ∪ {t::nat. t0 ≤ t ∧ t < t0+n ∧ c' t}⟩
    using merge_def by auto
  hence ⟨card {t::nat. t0 ≤ t ∧ t < t0+n ∧ ((c⊕c') t)}
          ≤ card {t::nat. t0 ≤ t ∧ t < t0+n ∧ c t}
            + card {t::nat. t0 ≤ t ∧ t < t0+n ∧ 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 ≡ ∀t. tick_count c t n ≤ 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
```

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

```
lemma spor_bounded:
  assumes ‹p_sporadic n c›
    shows ‹bounded (Suc n) 1 c›
proof -
  from assms have ‹∀t. c t ⟶ (∀t'. (t < t' ∧ t' ≤ t+n) ⟶ ¬(c t'))›
```

```
    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 .
qed
```

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

```
lemma spor_bounded2:
  assumes ‹p_sporadic n c›
    shows ‹bounded (Suc (Suc n)) 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)
qed
```

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

```
lemma bounded_less:
  assumes ‹bounded n' m c›
      and ‹n' ≥ 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' ≥ n›
    shows ‹bounded n (m+m') (c⊕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' ≥ n›
    shows ‹bounded (Suc n) 2 (c⊕c')›
proof -
  have 1:‹bounded (Suc n) 1 c› using spor_bounded[OF assms(1)] .
  have 2:‹bounded (Suc n') 1 c'› using spor_bounded[OF assms(2)] .
  from assms(3) have 3:‹Suc n' ≥ Suc n› by simp
  have ‹1+1 = (2::nat)› by simp
  with bounded_merge[OF 1 2 3] show ?thesis by metis
qed
```

5 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 (Suc (min n n')) 2 (c⊕c')›
using assms bounded_less bounded_merge sporadic_bounded1 spor_bounded
by (metis (no_types, lifting) min.cobounded1 min_Suc_Suc min_def one_add_one)
```

6 Tests

```
abbreviation ‹c1::clock ≡ (λt. t ≥ 1 ∧ (t-1) mod 2 = 0)›
abbreviation ‹c2::clock ≡ (λt. t ≥ 2 ∧ (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 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 ‹t0 mod 2 = 0› by arith
              hence ‹{t. (t = t0 ∨ t = t0 + 1) ∧ 1 ≤ t ∧ (t - 1) mod 2 = 0} = {t0+1}›
                by auto
              hence ‹{t. t0 ≤ t ∧ t < t0 + 2 ∧ 1 ≤ t ∧ (t - 1) mod 2 = 0} = {t0+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
```