Timed Automata

Simon Wimmer

May 26, 2024

Abstract

Timed automata are a widely used formalism for modeling real-time systems, which is employed in a class of successful model checkers such as UPPAAL [LPY97], HyTech [HHWt97] or Kronos [Yov97]. This work formalizes the theory for the subclass of diagonal-free timed automata, which is sufficient to model many interesting problems. We first define the basic concepts and semantics of diagonal-free timed automata. Based on this, we prove two types of decidability results for the language emptiness problem.

The first is the classic result of Alur and Dill [AD90, AD94], which uses a finite partitioning of the state space into so-called *regions*.

Our second result focuses on an approach based on *Difference Bound Matrices (DBMs)*, which is practically used by model checkers. We prove the correctness of the basic forward analysis operations on DBMs. One of these operations is the Floyd-Warshall algorithm for the all-pairs shortest paths problem. To obtain a finite search space, a widening operation has to be used for this kind of analysis. We use Patricia Bouyer's [Bou04] approach to prove that this widening operation is correct in the sense that DBM-based forward analysis in combination with the widening operation also decides language emptiness. The interesting property of this proof is that the first decidability result is reused to obtain the second one.

Contents

theory *Floyd-Warshall*
imports *Main*
begin

1 Floyd-Warshall Algorithm for the All-Pairs Shortest Paths Problem

Auxiliary lemma *distinct-list-single-elem-decomp*: $\{xs.\ set\ xs \subseteq \{0\} \wedge distinct\ xs\} = \{[], [0]\}$
⟨*proof*⟩

1.1 Cycles in Lists

abbreviation $cnt\ x\ xs \equiv length\ (filter\ (\lambda y.\ x = y)\ xs)$

fun *remove-cycles* :: $'a\ list \Rightarrow 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$
where
remove-cycles [] - *acc* = *rev acc* |
remove-cycles (*x*#*xs*) *y acc* =
(*if x* = *y then remove-cycles xs y* [*x*] *else remove-cycles xs y* (*x*#*acc*))

lemma *cnt-rev*: *cnt x* (*rev xs*) = *cnt x xs* ⟨*proof*⟩

value *as* @ [*x*] @ *bs* @ [*x*] @ *cs* @ [*x*] @ *ds*

lemma *remove-cycles-removes*: *cnt x* (*remove-cycles xs x ys*) ≤ *max 1* (*cnt x ys*)
⟨*proof*⟩

lemma *remove-cycles-id*: $x \notin set\ xs \Longrightarrow remove\text{-}cycles\ xs\ x\ ys = rev\ ys\ @\ xs$
⟨*proof*⟩

lemma *remove-cycles-cnt-id*:
$x \neq y \Longrightarrow cnt\ y\ (remove\text{-}cycles\ xs\ x\ ys) \leq cnt\ y\ ys + cnt\ y\ xs$
⟨*proof*⟩

lemma *remove-cycles-ends-cycle*: $remove\text{-}cycles\ xs\ x\ ys \neq rev\ ys\ @\ xs \Longrightarrow x \in set\ xs$
⟨*proof*⟩

lemma *remove-cycles-begins-with*: $x \in set\ xs \Longrightarrow \exists\ zs.\ remove\text{-}cycles\ xs\ x\ ys = x\ \#\ zs \wedge x \notin set\ zs$

⟨*proof*⟩

lemma *remove-cycles-self*:
x ∈ *set xs* ⟹ *remove-cycles* (*remove-cycles xs x ys*) *x zs* = *remove-cycles xs x ys*
⟨*proof*⟩

lemma *remove-cycles-one*: *remove-cycles* (*as* @ *x* # *xs*) *x ys* = *remove-cycles* (*x*#*xs*) *x ys*
⟨*proof*⟩

lemma *remove-cycles-cycles*:
x ∈ *set xs* ⟹ ∃ *xxs as*. *as* @ *concat* (*map* (λ *xs*. *x* # *xs*) *xxs*) @ *remove-cycles xs x ys* = *xs* ∧ *x* ∉ *set as*
⟨*proof*⟩

fun *start-remove* :: *'a list* ⇒ *'a* ⇒ *'a list* ⇒ *'a list*
where
start-remove [] - *acc* = *rev acc* |
start-remove (*x*#*xs*) *y acc* =
(*if x* = *y then rev acc* @ *remove-cycles xs y* [*y*] *else start-remove xs y* (*x* # *acc*))

lemma *start-remove-decomp*:
x ∈ *set xs* ⟹ ∃ *as bs*. *xs* = *as* @ *x* # *bs* ∧ *start-remove xs x ys* = *rev ys* @ *as* @ *remove-cycles bs x* [*x*]
⟨*proof*⟩

lemma *start-remove-removes*: *cnt x* (*start-remove xs x ys*) ≤ *Suc* (*cnt x ys*)
⟨*proof*⟩

lemma *start-remove-id*[*simp*]: *x* ∉ *set xs* ⟹ *start-remove xs x ys* = *rev ys* @ *xs*
⟨*proof*⟩

lemma *start-remove-cnt-id*:
x ≠ *y* ⟹ *cnt y* (*start-remove xs x ys*) ≤ *cnt y ys* + *cnt y xs*
⟨*proof*⟩

fun *remove-all-cycles* :: *'a list* ⇒ *'a list* ⇒ *'a list*
where
remove-all-cycles [] *xs* = *xs* |
remove-all-cycles (*x* # *xs*) *ys* = *remove-all-cycles xs* (*start-remove ys x* [])

lemma *cnt-remove-all-mono*:*cnt y* (*remove-all-cycles xs ys*) ≤ *max 1* (*cnt y ys*)
⟨*proof*⟩

lemma *cnt-remove-all-cycles*: *x* ∈ *set xs* ⟹ *cnt x* (*remove-all-cycles xs ys*) ≤ *1*
⟨*proof*⟩

lemma *cnt-mono*:
cnt a (*b* # *xs*) ≤ *cnt a* (*b* # *c* # *xs*)
⟨*proof*⟩

lemma *cnt-distinct-intro*: ∀ *x* ∈ *set xs*. *cnt x xs* ≤ *1* ⟹ *distinct xs*
⟨*proof*⟩

lemma *remove-cycles-subs*:
set (*remove-cycles xs x ys*) ⊆ *set xs* ∪ *set ys*
⟨*proof*⟩

lemma *start-remove-subs*:
set (*start-remove xs x ys*) ⊆ *set xs* ∪ *set ys*
⟨*proof*⟩

lemma *remove-all-cycles-subs*:
set (*remove-all-cycles xs ys*) ⊆ *set ys*
⟨*proof*⟩

lemma *remove-all-cycles-distinct*: *set ys* ⊆ *set xs* ⟹ *distinct* (*remove-all-cycles xs ys*)
⟨*proof*⟩

lemma *distinct-remove-cycles-inv*: *distinct* (*xs* @ *ys*) ⟹ *distinct* (*remove-cycles xs x ys*)
⟨*proof*⟩

definition *remove-all x xs* = (*if x* ∈ *set xs then tl* (*remove-cycles xs x* []) *else xs*)

definition *remove-all-rev x xs* = (*if x* ∈ *set xs then rev* (*tl* (*remove-cycles* (*rev xs*) *x* [])) *else xs*)

lemma *remove-all-distinct*:
distinct xs ⟹ *distinct* (*x* # *remove-all x xs*)

⟨*proof*⟩

lemma *remove-all-removes*:
  $x \notin$ *set* (*remove-all x xs*)
⟨*proof*⟩

lemma *remove-all-subs*:
  set (*remove-all x xs*) $\subseteq$ *set xs*
⟨*proof*⟩

lemma *remove-all-rev-distinct*: *distinct xs* $\Longrightarrow$ *distinct* (*x # remove-all-rev x xs*)
⟨*proof*⟩

lemma *remove-all-rev-removes*: $x \notin$ *set* (*remove-all-rev x xs*)
⟨*proof*⟩

lemma *remove-all-rev-subs*: *set* (*remove-all-rev x xs*) $\subseteq$ *set xs*
⟨*proof*⟩

abbreviation *rem-cycles i j xs* $\equiv$ *remove-all i* (*remove-all-rev j* (*remove-all-cycles xs xs*))

lemma *rem-cycles-distinct'*: $i \neq j \Longrightarrow$ *distinct* (*i # j # rem-cycles i j xs*)
⟨*proof*⟩

lemma *rem-cycles-removes-last*: $j \notin$ *set* (*rem-cycles i j xs*)
⟨*proof*⟩

lemma *rem-cycles-distinct*: *distinct* (*rem-cycles i j xs*)
⟨*proof*⟩

lemma *rem-cycles-subs*: *set* (*rem-cycles i j xs*) $\subseteq$ *set xs*
⟨*proof*⟩

1.2 Definition of the Algorithm

We formalize the Floyd-Warshall algorithm on a linearly ordered abelian semigroup. However, we would not need an *abelian* monoid if we had the right type class.

class *linordered-ab-monoid-add* = *linordered-ab-semigroup-add* +
  fixes *neutral* :: $'a$ (**1**)
    assumes *neutl*[*simp*]: **1** + $x = x$

assumes *neutr*[*simp*]: $x + \mathbf{1} = x$
begin

lemmas *assoc* = *add.assoc*

type-synonym $'c\ mat = nat \Rightarrow nat \Rightarrow 'c$

definition (**in** −) *upd* :: $'c\ mat \Rightarrow nat \Rightarrow nat \Rightarrow 'c \Rightarrow 'c\ mat$
where
upd m x y v = *m* (*x* := (*m x*) (*y* := *v*))

definition *fw-upd* :: $'a\ mat \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow 'a\ mat$ **where**
fw-upd m k i j ≡ *upd m i j* (*min* (*m i j*) (*m i k* + *m k j*))

lemma *fw-upd-mono*:
fw-upd m k i j i' j' ≤ *m i' j'*
⟨*proof*⟩

fun *fw* :: $'a\ mat \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow 'a\ mat$ **where**
fw m n 0 0 0 = *fw-upd m 0 0 0* |
fw m n (*Suc k*) *0 0* = *fw-upd* (*fw m n k n n*) (*Suc k*) *0 0* |
fw m n k (*Suc i*) *0* = *fw-upd* (*fw m n k i n*) *k* (*Suc i*) *0* |
fw m n k i (*Suc j*) = *fw-upd* (*fw m n k i j*) *k i* (*Suc j*)

lemma *fw-invariant-aux-1*:
$j'' \leq j \Longrightarrow i \leq n \Longrightarrow j \leq n \Longrightarrow k \leq n \Longrightarrow$ *fw m n k i j i' j'* ≤ *fw m n k i j'' i' j'*
⟨*proof*⟩

lemma *fw-invariant-aux-2*:
$i \leq n \Longrightarrow j \leq n \Longrightarrow k \leq n \Longrightarrow i'' \leq i \Longrightarrow j'' \leq j$
$\Longrightarrow$ *fw m n k i j i' j'* ≤ *fw m n k i'' j'' i' j'*
⟨*proof*⟩

lemma *fw-invariant*:
$k' \leq k \Longrightarrow i \leq n \Longrightarrow j \leq n \Longrightarrow k \leq n \Longrightarrow j'' \leq j \Longrightarrow i'' \leq i$
$\Longrightarrow$ *fw m n k i j i' j'* ≤ *fw m n k' i'' j'' i' j'*
⟨*proof*⟩

lemma *single-row-inv*:
$j' < j \Longrightarrow j \leq n \Longrightarrow i' \leq n \Longrightarrow$ *fw m n k i' j i' j'* = *fw m n k i' j' i' j'*
⟨*proof*⟩

lemma *single-iteration-inv'*:

$i' < i \Longrightarrow j' \leq n \Longrightarrow j \leq n \Longrightarrow i \leq n \Longrightarrow fw\ m\ n\ k\ i\ j\ i'\ j' = fw\ m\ n\ k\ i'\ j'\ i'\ j'$
⟨*proof*⟩

lemma *single-iteration-inv*:
$i' \leq i \Longrightarrow j' \leq j \Longrightarrow i \leq n \Longrightarrow j \leq n \Longrightarrow fw\ m\ n\ k\ i\ j\ i'\ j' = fw\ m\ n\ k\ i'\ j'\ i'\ j'$
⟨*proof*⟩

lemma *fw-innermost-id*:
$i \leq n \Longrightarrow j \leq n \Longrightarrow j' \leq n \Longrightarrow i' < i \Longrightarrow fw\ m\ n\ 0\ i'\ j'\ i\ j = m\ i\ j$
⟨*proof*⟩

lemma *fw-middle-id*:
$i \leq n \Longrightarrow j \leq n \Longrightarrow j' < j \Longrightarrow i' \leq i \Longrightarrow fw\ m\ n\ 0\ i'\ j'\ i\ j = m\ i\ j$
⟨*proof*⟩

lemma *fw-outermost-mono*:
$i \leq n \Longrightarrow j \leq n \Longrightarrow fw\ m\ n\ 0\ i\ j\ i\ j \leq m\ i\ j$
⟨*proof*⟩

lemma *Suc-innermost-id1*:
$i \leq n \Longrightarrow j \leq n \Longrightarrow j' \leq n \Longrightarrow i' < i \Longrightarrow fw\ m\ n\ (Suc\ k)\ i'\ j'\ i\ j = fw\ m\ n\ k\ i\ j\ i\ j$
⟨*proof*⟩

lemma *Suc-innermost-id2*:
$i \leq n \Longrightarrow j \leq n \Longrightarrow j' < j \Longrightarrow i' \leq i \Longrightarrow fw\ m\ n\ (Suc\ k)\ i'\ j'\ i\ j = fw\ m\ n\ k\ i\ j\ i\ j$
⟨*proof*⟩

lemma *Suc-innermost-id1'*:
$i \leq n \Longrightarrow j \leq n \Longrightarrow j' \leq n \Longrightarrow i' < i \Longrightarrow fw\ m\ n\ (Suc\ k)\ i'\ j'\ i\ j = fw\ m\ n\ k\ n\ n\ i\ j$
⟨*proof*⟩

lemma *Suc-innermost-id2'*:
$i \leq n \Longrightarrow j \leq n \Longrightarrow j' < j \Longrightarrow i' \leq i \Longrightarrow fw\ m\ n\ (Suc\ k)\ i'\ j'\ i\ j = fw\ m\ n\ k\ n\ n\ i\ j$
⟨*proof*⟩

lemma *Suc-innermost-mono*:
$i \leq n \Longrightarrow j \leq n \Longrightarrow fw\ m\ n\ (Suc\ k)\ i\ j\ i\ j \leq fw\ m\ n\ k\ i\ j\ i\ j$
⟨*proof*⟩

lemma *fw-mono'*:
$i \leq n \Longrightarrow j \leq n \Longrightarrow fw\ m\ n\ k\ i\ j\ i\ j \leq m\ i\ j$
⟨*proof*⟩

lemma *fw-mono*:
$i \leq n \Longrightarrow j \leq n \Longrightarrow i' \leq n \Longrightarrow j' \leq n \Longrightarrow fw\ m\ n\ k\ i\ j\ i'\ j' \leq m\ i'\ j'$
⟨*proof*⟩

lemma *add-mono-neutr*:
assumes $\mathbf{1} \leq b$
shows $a \leq a + b$
⟨*proof*⟩

lemma *add-mono-neutl*:
assumes $\mathbf{1} \leq b$
shows $a \leq b + a$
⟨*proof*⟩

lemma *fw-step-0*:
$m\ 0\ 0 \geq \mathbf{1} \Longrightarrow i \leq n \Longrightarrow j \leq n \Longrightarrow fw\ m\ n\ 0\ i\ j\ i\ j = min\ (m\ i\ j)\ (m\ i\ 0 + m\ 0\ j)$
⟨*proof*⟩

lemma *fw-step-Suc*:
$\forall\ k'{\leq}n.\ fw\ m\ n\ k\ n\ n\ k'\ k' \geq \mathbf{1} \Longrightarrow i \leq n \Longrightarrow j \leq n \Longrightarrow Suc\ k \leq n$
$\Longrightarrow fw\ m\ n\ (Suc\ k)\ i\ j\ i\ j = min\ (fw\ m\ n\ k\ n\ n\ i\ j)\ (fw\ m\ n\ k\ n\ n\ i\ (Suc\ k) + fw\ m\ n\ k\ n\ n\ (Suc\ k)\ j)$
⟨*proof*⟩

1.2.1 Length of Paths

fun *len* :: $'a\ mat \Rightarrow nat \Rightarrow nat \Rightarrow nat\ list \Rightarrow 'a$ **where**
len m u v [] = *m u v* |
len m u v (*w*#*ws*) = *m u w* + *len m w v ws*

lemma *len-decomp*: $xs = ys\ @\ y\ \#\ zs \Longrightarrow len\ m\ x\ z\ xs = len\ m\ x\ y\ ys + len\ m\ y\ z\ zs$
⟨*proof*⟩

lemma *len-comp*: *len m a c* (*xs* @ *b* # *ys*) = *len m a b xs* + *len m b c ys*
⟨*proof*⟩

1.2.2 Shortening Negative Cycles

lemma *remove-cycles-neg-cycles-aux*:
fixes i xs ys
defines $xs' \equiv i \# ys$
assumes $i \notin set\ ys$
assumes $i \in set\ xs$
assumes $xs = as\ @\ concat\ (map\ ((\#)\ i)\ xss)\ @\ xs'$
assumes $len\ m\ i\ j\ ys > len\ m\ i\ j\ xs$
shows $\exists\ ys.\ set\ ys \subseteq set\ xs \wedge len\ m\ i\ i\ ys < \mathbf{1}$ ⟨*proof*⟩

lemma *add-lt-neutral*: $a + b < b \Longrightarrow a < \mathbf{1}$
⟨*proof*⟩

lemma *remove-cycles-neg-cycles-aux'*:
fixes j xs ys
assumes $j \notin set\ ys$
assumes $j \in set\ xs$
assumes $xs = ys\ @\ j \#\ concat\ (map\ (\lambda\ xs.\ xs\ @\ [j])\ xss)\ @\ as$
assumes $len\ m\ i\ j\ ys > len\ m\ i\ j\ xs$
shows $\exists\ ys.\ set\ ys \subseteq set\ xs \wedge len\ m\ j\ j\ ys < \mathbf{1}$ ⟨*proof*⟩

lemma *add-le-impl*: $a + b < a + c \Longrightarrow b < c$
⟨*proof*⟩

lemma *start-remove-neg-cycles*:
$len\ m\ i\ j\ (start\text{-}remove\ xs\ k\ []) > len\ m\ i\ j\ xs \Longrightarrow \exists\ ys.\ set\ ys \subseteq set\ xs \wedge len\ m\ k\ k\ ys < \mathbf{1}$
⟨*proof*⟩

lemma *remove-all-cycles-neg-cycles*:
$len\ m\ i\ j\ (remove\text{-}all\text{-}cycles\ ys\ xs) > len\ m\ i\ j\ xs$
$\Longrightarrow \exists\ ys\ k.\ set\ ys \subseteq set\ xs \wedge k \in set\ xs \wedge len\ m\ k\ k\ ys < \mathbf{1}$
⟨*proof*⟩

lemma (**in** −) *concat-map-cons-rev*:
$rev\ (concat\ (map\ ((\#)\ j)\ xss)) = concat\ (map\ (\lambda\ xs.\ xs\ @\ [j])\ (rev\ (map\ rev\ xss)))$
⟨*proof*⟩

lemma *negative-cycle-dest*: $len\ m\ i\ j\ (rem\text{-}cycles\ i\ j\ xs) > len\ m\ i\ j\ xs$
$\Longrightarrow \exists\ i'\ ys.\ len\ m\ i'\ i'\ ys < \mathbf{1} \wedge set\ ys \subseteq set\ xs \wedge i' \in set\ (i \#\ j \#\ xs)$
⟨*proof*⟩

1.3 Definition of Shortest Paths

definition $D :: 'a\ mat \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow 'a$ **where**
$D\ m\ i\ j\ k \equiv Min\ \{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..k\} \wedge i \notin set\ xs \wedge j \notin set\ xs \wedge distinct\ xs\}$

lemma (**in** −) *distinct-length-le*:*finite* $s \Longrightarrow set\ xs \subseteq s \Longrightarrow distinct\ xs \Longrightarrow length\ xs \leq card\ s$
⟨*proof*⟩

lemma *D-base-finite*:
$finite\ \{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..k\} \wedge distinct\ xs\}$
⟨*proof*⟩

lemma *D-base-finite*′:
$finite\ \{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..k\} \wedge distinct\ (i\ \#\ j\ \#\ xs)\}$
⟨*proof*⟩

lemma *D-base-finite*″:
$finite\ \{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..k\} \wedge i \notin set\ xs \wedge j \notin set\ xs \wedge distinct\ xs\}$
⟨*proof*⟩

definition *cycle-free* $:: 'a\ mat \Rightarrow nat \Rightarrow bool$ **where**
$cycle\text{-}free\ m\ n \equiv \forall\ i\ xs.\ i \leq n \wedge set\ xs \subseteq \{0..n\} \longrightarrow$
$(\forall\ j.\ j \leq n \longrightarrow len\ m\ i\ j\ (rem\text{-}cycles\ i\ j\ xs) \leq len\ m\ i\ j\ xs) \wedge len\ m\ i\ i\ xs \geq \mathbf{1}$

lemma *D-eqI*:
fixes $m\ n\ i\ j\ k$
defines $A \equiv \{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..k\}\}$
defines $A\text{-}distinct \equiv \{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..k\} \wedge i \notin set\ xs \wedge j \notin set\ xs \wedge distinct\ xs\}$
assumes $cycle\text{-}free\ m\ n\ i \leq n\ j \leq n\ k \leq n\ (\bigwedge y.\ y \in A\text{-}distinct \Longrightarrow x \leq y)\ x \in A$
shows $D\ m\ i\ j\ k = x$ ⟨*proof*⟩

lemma *D-base-not-empty*:
$\{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..k\} \wedge i \notin set\ xs \wedge j \notin set\ xs \wedge distinct\ xs\} \neq \{\}$
⟨*proof*⟩

lemma *Min-elem-dest*: $finite\ A \Longrightarrow A \neq \{\} \Longrightarrow x = Min\ A \Longrightarrow x \in A$
⟨*proof*⟩

lemma *D-dest*: $x = D\ m\ i\ j\ k \Longrightarrow$
$x \in \{len\ m\ i\ j\ xs\ |xs.\ set\ xs \subseteq \{0..Suc\ k\} \wedge i \notin set\ xs \wedge j \notin set\ xs \wedge distinct\ xs\}$
⟨*proof*⟩

lemma *D-dest'*: $x = D\ m\ i\ j\ k \Longrightarrow x \in \{len\ m\ i\ j\ xs\ |xs.\ set\ xs \subseteq \{0..Suc\ k\}\}$
⟨*proof*⟩

lemma *D-dest''*: $x = D\ m\ i\ j\ k \Longrightarrow x \in \{len\ m\ i\ j\ xs\ |xs.\ set\ xs \subseteq \{0..k\}\}$
⟨*proof*⟩

lemma *cycle-free-loop-dest*: $i \leq n \Longrightarrow set\ xs \subseteq \{0..n\} \Longrightarrow cycle\text{-}free\ m\ n \Longrightarrow len\ m\ i\ i\ xs \geq \mathbf{1}$
⟨*proof*⟩

lemma *cycle-free-dest*:
$cycle\text{-}free\ m\ n \Longrightarrow i \leq n \Longrightarrow j \leq n \Longrightarrow set\ xs \subseteq \{0..n\}$
$\Longrightarrow len\ m\ i\ j\ (rem\text{-}cycles\ i\ j\ xs) \leq len\ m\ i\ j\ xs$
⟨*proof*⟩

definition *cycle-free-up-to* :: $'a\ mat \Rightarrow nat \Rightarrow nat \Rightarrow bool$ **where**
$cycle\text{-}free\text{-}up\text{-}to\ m\ k\ n \equiv \forall\ i\ xs.\ i \leq n \wedge set\ xs \subseteq \{0..k\} \longrightarrow$
$(\forall\ j.\ j \leq n \longrightarrow len\ m\ i\ j\ (rem\text{-}cycles\ i\ j\ xs) \leq len\ m\ i\ j\ xs) \wedge len\ m\ i\ i\ xs \geq \mathbf{1}$

lemma *cycle-free-up-to-loop-dest*:
$i \leq n \Longrightarrow set\ xs \subseteq \{0..k\} \Longrightarrow cycle\text{-}free\text{-}up\text{-}to\ m\ k\ n \Longrightarrow len\ m\ i\ i\ xs \geq \mathbf{1}$
⟨*proof*⟩

lemma *cycle-free-up-to-diag*:
assumes $cycle\text{-}free\text{-}up\text{-}to\ m\ k\ n\ i \leq n$
shows $m\ i\ i \geq \mathbf{1}$
⟨*proof*⟩

lemma *D-eqI2*:
fixes $m\ n\ i\ j\ k$
defines $A \equiv \{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..k\}\}$
defines $A\text{-}distinct \equiv \{len\ m\ i\ j\ xs \mid xs.\ set\ xs \subseteq \{0..k\} \wedge i \notin set\ xs \wedge j \notin set\ xs \wedge distinct\ xs\}$
assumes $cycle\text{-}free\text{-}up\text{-}to\ m\ k\ n\ i \leq n\ j \leq n\ k \leq n$
$(\bigwedge y.\ y \in A\text{-}distinct \Longrightarrow x \leq y)\ x \in A$
shows $D\ m\ i\ j\ k = x$ ⟨*proof*⟩

1.4 Result Under The Absence of Negative Cycles

This proves that the algorithm correctly computes shortest paths under the absence of negative cycles by a standard argument.

theorem *fw-shortest-path-up-to*:
cycle-free-up-to m k n $\Longrightarrow i' \leq i \Longrightarrow j' \leq j \Longrightarrow i \leq n \Longrightarrow j \leq n \Longrightarrow k \leq n$
$\Longrightarrow D\ m\ i'\ j'\ k = fw\ m\ n\ k\ i\ j\ i'\ j'$
⟨*proof*⟩

lemma *cycle-free-cycle-free-up-to*:
cycle-free m n $\Longrightarrow k \leq n \Longrightarrow$ *cycle-free-up-to m k n*
⟨*proof*⟩

lemma *cycle-free-diag*:
cycle-free m n $\Longrightarrow i \leq n \Longrightarrow \mathbf{1} \leq m\ i\ i$
⟨*proof*⟩

corollary *fw-shortest-path*:
cycle-free m n $\Longrightarrow i' \leq i \Longrightarrow j' \leq j \Longrightarrow i \leq n \Longrightarrow j \leq n \Longrightarrow k \leq n$
$\Longrightarrow D\ m\ i'\ j'\ k = fw\ m\ n\ k\ i\ j\ i'\ j'$
⟨*proof*⟩

corollary *fw-shortest*:
assumes *cycle-free m n* $i \leq n\ j \leq n\ k \leq n$
shows $fw\ m\ n\ n\ n\ n\ i\ j \leq fw\ m\ n\ n\ n\ n\ i\ k + fw\ m\ n\ n\ n\ n\ k\ j$
⟨*proof*⟩

1.5 Result Under the Presence of Negative Cycles

lemma *not-cylce-free-dest*: ¬ *cycle-free m n* $\Longrightarrow \exists\ k \leq n.$ ¬ *cycle-free-up-to m k n*
⟨*proof*⟩

lemma *D-not-diag-le*:
$(x :: 'a) \in \{len\ m\ i\ j\ xs\ |xs.\ set\ xs \subseteq \{0..k\} \wedge i \notin set\ xs \wedge j \notin set\ xs \wedge distinct\ xs\}$
$\Longrightarrow D\ m\ i\ j\ k \leq x$ ⟨*proof*⟩

lemma *D-not-diag-le'*: *set xs* $\subseteq \{0..k\} \Longrightarrow i \notin set\ xs \Longrightarrow j \notin set\ xs \Longrightarrow distinct\ xs$
$\Longrightarrow D\ m\ i\ j\ k \leq len\ m\ i\ j\ xs$ ⟨*proof*⟩

lemma (**in** −) *nat-upto-subs-top-removal'*:

$S \subseteq \{0..Suc\ n\} \Longrightarrow Suc\ n \notin S \Longrightarrow S \subseteq \{0..n\}$
⟨*proof*⟩

lemma (**in** −) *nat-upto-subs-top-removal*:
$S \subseteq \{0..n::nat\} \Longrightarrow n \notin S \Longrightarrow S \subseteq \{0..n - 1\}$
⟨*proof*⟩

lemma *fw-Suc*:
$i \leq n \Longrightarrow j \leq n \Longrightarrow i' \leq n \Longrightarrow j' \leq n \Longrightarrow fw\ m\ n\ (Suc\ k)\ i'\ j'\ i\ j \leq fw\ m\ n\ k\ n\ n\ i\ j$
⟨*proof*⟩

lemma *negative-len-shortest*:
$length\ xs = n \Longrightarrow len\ m\ i\ i\ xs < \mathbf{1}$
$\Longrightarrow \exists\ j\ ys.\ distinct\ (j\ \#\ ys) \wedge len\ m\ j\ j\ ys < \mathbf{1} \wedge j \in set\ (i\ \#\ xs) \wedge set\ ys \subseteq set\ xs$
⟨*proof*⟩

theorem *FW-neg-cycle-detect*:
$\neg\ cycle\text{-}free\ m\ n \Longrightarrow \exists\ i \leq n.\ fw\ m\ n\ n\ n\ n\ i\ i < \mathbf{1}$
⟨*proof*⟩

end
end
theory *Timed-Automata*
imports *Main*
begin

2 Basic Definitions and Semantics

2.1 Time

class *time* = *linordered-ab-group-add* +
assumes *dense*: $x < y \Longrightarrow \exists z.\ x < z \wedge z < y$
assumes *non-trivial*: $\exists\ x.\ x \neq 0$

begin

lemma *non-trivial-neg*: $\exists\ x.\ x < 0$
⟨*proof*⟩

end

datatype ('*c*, '*t* :: *time*) *cconstraint* =

AND ('*c*, '*t*) *cconstraint* ('*c*, '*t*) *cconstraint* |
LT '*c* '*t* |
LE '*c* '*t* |
EQ '*c* '*t* |
GT '*c* '*t* |
GE '*c* '*t*

2.2 Syntactic Definition

For an informal description of timed automata we refer to Bengtsson and Yi [BY03]. We define a timed automaton *A*

type-synonym
('*c*, '*time*, '*s*) *invassn* = '*s* ⇒ ('*c*, '*time*) *cconstraint*

type-synonym
('*a*, '*c*, '*time*, '*s*) *transition* = '*s* ∗ ('*c*, '*time*) *cconstraint* ∗ '*a* ∗ '*c list* ∗ '*s*

type-synonym
('*a*, '*c*, '*time*, '*s*) *ta* = ('*a*, '*c*, '*time*, '*s*) *transition set* ∗ ('*c*, '*time*, '*s*) *invassn*

definition *trans-of* :: ('*a*, '*c*, '*time*, '*s*) *ta* ⇒ ('*a*, '*c*, '*time*, '*s*) *transition set*
where
trans-of ≡ *fst*
definition *inv-of* :: ('*a*, '*c*, '*time*, '*s*) *ta* ⇒ ('*c*, '*time*, '*s*) *invassn* **where**
inv-of ≡ *snd*

abbreviation *transition* ::
('*a*, '*c*, '*time*, '*s*) *ta* ⇒ '*s* ⇒ ('*c*, '*time*) *cconstraint* ⇒ '*a* ⇒ '*c list* ⇒ '*s* ⇒ *bool*
(- ⊢ - ⟶^-,-,- - [*61*,*61*,*61*,*61*,*61*,*61*] *61*) **where**
(*A* ⊢ *l* ⟶^*g*,*a*,*r* *l*′) ≡ (*l*,*g*,*a*,*r*,*l*′) ∈ *trans-of A*

2.2.1 Collecting Information About Clocks

fun *collect-clks* :: ('*c*, '*t* :: *time*) *cconstraint* ⇒ '*c set*
where
collect-clks (*AND cc1 cc2*) = *collect-clks cc1* ∪ *collect-clks cc2* |
collect-clks (*LT c* -) = {*c*} |
collect-clks (*LE c* -) = {*c*} |
collect-clks (*EQ c* -) = {*c*} |
collect-clks (*GE c* -) = {*c*} |
collect-clks (*GT c* -) = {*c*}

fun *collect-clock-pairs* :: ('*c*, '*t* :: *time*) *cconstraint* ⇒ ('*c* ∗ '*t*) *set*
where
collect-clock-pairs (*LT x m*) = {(*x*, *m*)} |
collect-clock-pairs (*LE x m*) = {(*x*, *m*)} |
collect-clock-pairs (*EQ x m*) = {(*x*, *m*)} |
collect-clock-pairs (*GE x m*) = {(*x*, *m*)} |
collect-clock-pairs (*GT x m*) = {(*x*, *m*)} |
collect-clock-pairs (*AND cc1 cc2*) = (*collect-clock-pairs cc1* ∪ *collect-clock-pairs cc2*)

definition *collect-clkt* :: ('*a*, '*c*, '*t*::*time*, '*s*) *transition set* ⇒ ('*c* ∗'*t*) *set*
where
collect-clkt S = ⋃ {*collect-clock-pairs* (*fst* (*snd t*)) | *t* . *t* ∈ *S*}

definition *collect-clki* :: ('*c*, '*t* :: *time*, '*s*) *invassn* ⇒ ('*c* ∗'*t*) *set*
where
collect-clki I = ⋃ {*collect-clock-pairs* (*I x*) | *x*. *True*}

definition *clkp-set* :: ('*a*, '*c*, '*t* :: *time*, '*s*) *ta* ⇒ ('*c* ∗'*t*) *set*
where
clkp-set A = *collect-clki* (*inv-of A*) ∪ *collect-clkt* (*trans-of A*)

definition *collect-clkvt* :: ('*a*, '*c*, '*t*::*time*, '*s*) *transition set* ⇒ '*c set*
where
collect-clkvt S = ⋃ {*set* ((*fst o snd o snd o snd*) *t*) | *t* . *t* ∈ *S*}

abbreviation *clk-set* **where** *clk-set A* ≡ *fst* ‘ *clkp-set A* ∪ *collect-clkvt* (*trans-of A*)

inductive *valid-abstraction*
where
⟦∀ (*x*,*m*) ∈ *clkp-set A*. *m* ≤ *k x* ∧ *x* ∈ *X* ∧ *m* ∈ ℕ; *collect-clkvt* (*trans-of A*) ⊆ *X*; *finite X*⟧
⟹ *valid-abstraction A X k*

2.3 Operational Semantics

type-synonym ('*c*, '*t*) *cval* = '*c* ⇒ '*t*

definition *cval-add* :: ('*c*,'*t*) *cval* ⇒ '*t*::*time* ⇒ ('*c*,'*t*) *cval* (**infixr** ⊕ *64*)
where
u ⊕ *d* = (λ *x*. *u x* + *d*)

inductive *clock-val* :: (*'c*, *'t*) *cval* ⇒ (*'c*, *'t*::*time*) *cconstraint* ⇒ *bool* (- ⊢ - [*62*, *62*] *62*)
where
⟦*u* ⊢ *cc1*; *u* ⊢ *cc2*⟧ ⟹ *u* ⊢ *AND cc1 cc2* |
⟦*u c* < *d*⟧ ⟹ *u* ⊢ *LT c d* |
⟦*u c* ≤ *d*⟧ ⟹ *u* ⊢ *LE c d* |
⟦*u c* = *d*⟧ ⟹ *u* ⊢ *EQ c d* |
⟦*u c* ≥ *d*⟧ ⟹ *u* ⊢ *GE c d* |
⟦*u c* > *d*⟧ ⟹ *u* ⊢ *GT c d*

declare *clock-val.intros*[*intro*]

inductive-cases[*elim*!]: *u* ⊢ *AND cc1 cc2*
inductive-cases[*elim*!]: *u* ⊢ *LT c d*
inductive-cases[*elim*!]: *u* ⊢ *LE c d*
inductive-cases[*elim*!]: *u* ⊢ *EQ c d*
inductive-cases[*elim*!]: *u* ⊢ *GE c d*
inductive-cases[*elim*!]: *u* ⊢ *GT c d*

fun *clock-set* :: *'c list* ⇒ *'t*::*time* ⇒ (*'c*,*'t*) *cval* ⇒ (*'c*,*'t*) *cval*
where
clock-set [] - *u* = *u* |
clock-set (*c*#*cs*) *t u* = (*clock-set cs t u*)(*c*:=*t*)

abbreviation *clock-set-abbrv* :: *'c list* ⇒ *'t*::*time* ⇒ (*'c*,*'t*) *cval* ⇒ (*'c*,*'t*) *cval*
([-→-]- [*65*,*65*,*65*] *65*)
where
[*r* → *t*]*u* ≡ *clock-set r t u*

inductive *step-t* ::
(*'a*, *'c*, *'t*, *'s*) *ta* ⇒ *'s* ⇒ (*'c*, *'t*) *cval* ⇒ (*'t*::*time*) ⇒ *'s* ⇒ (*'c*, *'t*) *cval* ⇒ *bool*
(- ⊢ ⟨-, -⟩ →- ⟨-, -⟩ [*61*,*61*,*61*] *61*)
where
⟦*u* ⊢ *inv-of A l*; *u* ⊕ *d* ⊢ *inv-of A l*; *d* ≥ *0*⟧ ⟹ *A* ⊢ ⟨*l*, *u*⟩ →^{d} ⟨*l*, *u* ⊕ *d*⟩

declare *step-t.intros*[*intro*!]

inductive-cases[*elim*!]: *A* ⊢ ⟨*l*, *u*⟩ →^{d} ⟨*l'*,*u'*⟩

lemma *step-t-determinacy1*:
A ⊢ ⟨*l*, *u*⟩ →^{d} ⟨*l'*,*u'*⟩ ⟹ *A* ⊢ ⟨*l*, *u*⟩ →^{d} ⟨*l''*,*u''*⟩ ⟹ *l'* = *l''*
⟨*proof*⟩

lemma *step-t-determinacy2*:
$A \vdash \langle l,\ u\rangle \rightarrow^{d} \langle l',u'\rangle \Longrightarrow\ A \vdash \langle l,\ u\rangle \rightarrow^{d} \langle l'',u''\rangle \Longrightarrow u' = u''$
⟨*proof*⟩

lemma *step-t-cont1*:
$d \geq 0 \Longrightarrow e \geq 0 \Longrightarrow A \vdash \langle l,\ u\rangle \rightarrow^{d} \langle l',u'\rangle \Longrightarrow A \vdash \langle l',\ u'\rangle \rightarrow^{e} \langle l'',u''\rangle$
$\Longrightarrow A \vdash \langle l,\ u\rangle \rightarrow^{d+e} \langle l'',u''\rangle$
⟨*proof*⟩

inductive *step-a* ::
$('a,\ 'c,\ 't,\ 's)\ ta \Rightarrow 's \Rightarrow ('c,\ ('t::time))\ cval \Rightarrow 'a \Rightarrow 's \Rightarrow ('c,\ 't)\ cval \Rightarrow bool$
$(\text{-} \vdash \langle\text{-},\ \text{-}\rangle \rightarrow_{\text{-}} \langle\text{-},\ \text{-}\rangle\ [61,61,61]\ 61)$
where
$[\![A \vdash l \longrightarrow^{g,a,r} l';\ u \vdash g;\ u' \vdash inv\text{-}of\ A\ l';\ u' = [r \rightarrow 0]u]\!] \Longrightarrow (A \vdash \langle l,\ u\rangle \rightarrow_{a} \langle l',\ u'\rangle)$

inductive *step* ::
$('a,\ 'c,\ 't,\ 's)\ ta \Rightarrow 's \Rightarrow ('c,\ ('t::time))\ cval \Rightarrow 's \Rightarrow ('c,\ 't)\ cval \Rightarrow bool$
$(\text{-} \vdash \langle\text{-},\ \text{-}\rangle \rightarrow \langle\text{-},\text{-}\rangle\ [61,61,61]\ 61)$
where
step-a: $A \vdash \langle l,\ u\rangle \rightarrow_{a} \langle l',u'\rangle \Longrightarrow (A \vdash \langle l,\ u\rangle \rightarrow \langle l',u'\rangle)$ |
step-t: $A \vdash \langle l,\ u\rangle \rightarrow^{d} \langle l',u'\rangle \Longrightarrow (A \vdash \langle l,\ u\rangle \rightarrow \langle l',u'\rangle)$

inductive-cases[*elim*!]: $A \vdash \langle l,\ u\rangle \rightarrow \langle l',u'\rangle$

declare *step.intros*[*intro*]

inductive
steps :: $('a,\ 'c,\ 't,\ 's)\ ta \Rightarrow 's \Rightarrow ('c,\ ('t::time))\ cval \Rightarrow 's \Rightarrow ('c,\ 't)\ cval \Rightarrow bool$
$(\text{-} \vdash \langle\text{-},\ \text{-}\rangle \rightarrow* \langle\text{-},\ \text{-}\rangle\ [61,61,61]\ 61)$
where
refl: $A \vdash \langle l,\ u\rangle \rightarrow* \langle l,\ u\rangle$ |
step: $A \vdash \langle l,\ u\rangle \rightarrow \langle l',\ u'\rangle \Longrightarrow A \vdash \langle l',\ u'\rangle \rightarrow* \langle l'',\ u''\rangle \Longrightarrow A \vdash \langle l,\ u\rangle \rightarrow* \langle l'',\ u''\rangle$

declare *steps.intros*[*intro*]

2.4 Zone Semantics

type-synonym $('c,\ 't)\ zone = ('c,\ 't)\ cval\ set$

definition *zone-delay* :: $('c, ('t::time))\ zone \Rightarrow ('c, 't)\ zone$
$(\text{-}^{\uparrow}\ [71]\ 71)$
where
$Z^{\uparrow} = \{u \oplus d | u\ d.\ u \in Z \wedge d \geq (0::'t)\}$

definition *zone-set* :: $('c, 't::time)\ zone \Rightarrow 'c\ list \Rightarrow ('c, 't)\ zone$
$(\text{-}_{\text{-}} \rightarrow_0 [71]\ 71)$
where
$zone\text{-}set\ Z\ r = \{[r \rightarrow (0::'t)]u \mid u\ .\ u \in Z\}$

inductive *step-z* ::
$('a, 'c, 't, 's)\ ta \Rightarrow 's \Rightarrow ('c, ('t::time))\ zone \Rightarrow 's \Rightarrow ('c, 't)\ zone \Rightarrow bool$
$(\text{-} \vdash \langle \text{-}, \text{-} \rangle \rightsquigarrow \langle \text{-}, \text{-} \rangle\ [61,61,61]\ 61)$
where
step-t-z: $A \vdash \langle l, Z \rangle \rightsquigarrow \langle l, (Z \cap \{u.\ u \vdash inv\text{-}of\ A\ l\})^{\uparrow} \cap \{u.\ u \vdash inv\text{-}of\ A\ l\}\rangle$ |
step-a-z: $[\![A \vdash l \longrightarrow^{g,a,r} l']\!]$
$\Longrightarrow (A \vdash \langle l, Z \rangle \rightsquigarrow \langle l', zone\text{-}set\ (Z \cap \{u.\ u \vdash g\})\ r \cap \{u.\ u \vdash inv\text{-}of\ A\ l'\}\rangle\ )$

inductive-cases[*elim*!]: $A \vdash \langle l, u \rangle \rightsquigarrow \langle l', u' \rangle$

declare *step-z.intros*[*intro*]

lemma *step-z-sound*:
$A \vdash \langle l, Z \rangle \rightsquigarrow \langle l', Z' \rangle \Longrightarrow (\forall\ u' \in Z'.\ \exists\ u \in Z.\ A \vdash \langle l, u \rangle \rightarrow \langle l', u' \rangle)$
⟨*proof*⟩

lemma *step-z-complete*:
$A \vdash \langle l, u \rangle \rightarrow \langle l', u' \rangle \Longrightarrow u \in Z \Longrightarrow \exists\ Z'.\ A \vdash \langle l, Z \rangle \rightsquigarrow \langle l', Z' \rangle \wedge u' \in Z'$
⟨*proof*⟩

Corresponds to version in old papers – not strong enough for inductive proof over transitive closure relation.

lemma *step-z-complete1*:
$A \vdash \langle l, u \rangle \rightarrow \langle l', u' \rangle \Longrightarrow \exists\ Z.\ A \vdash \langle l, \{u\} \rangle \rightsquigarrow \langle l', Z \rangle \wedge u' \in Z$
⟨*proof*⟩

Easier proof.

lemma *step-z-complete2*:
$A \vdash \langle l, u \rangle \rightarrow \langle l', u' \rangle \Longrightarrow \exists\ Z.\ A \vdash \langle l, \{u\} \rangle \rightsquigarrow \langle l', Z \rangle \wedge u' \in Z$
⟨*proof*⟩

inductive

steps-z :: ('*a*, '*c*, '*t*, '*s*) *ta* ⇒ '*s* ⇒ ('*c*, ('*t*::*time*)) *zone* ⇒ '*s* ⇒ ('*c*, '*t*) *zone* ⇒ *bool*
(- ⊢ ⟨-, -⟩ ⇝* ⟨-, -⟩ [*61*,*61*,*61*] *61*)
where
refl: *A* ⊢ ⟨*l*, *Z*⟩ ⇝* ⟨*l*, *Z*⟩ |
step: *A* ⊢ ⟨*l*, *Z*⟩ ⇝ ⟨*l'*, *Z'*⟩ ⟹ *A* ⊢ ⟨*l'*, *Z'*⟩ ⇝* ⟨*l''*, *Z''*⟩ ⟹ *A* ⊢ ⟨*l*, *Z*⟩ ⇝* ⟨*l''*, *Z''*⟩

declare *steps-z.intros*[*intro*]

lemma *steps-z-sound*:
A ⊢ ⟨*l*, *Z*⟩ ⇝* ⟨*l'*, *Z'*⟩ ⟹ *u'* ∈ *Z'* ⟹ ∃ *u* ∈ *Z*. *A* ⊢ ⟨*l*, *u*⟩ →* ⟨*l'*, *u'*⟩
⟨*proof*⟩

lemma *steps-z-complete*:
A ⊢ ⟨*l*, *u*⟩ →* ⟨*l'*, *u'*⟩ ⟹ *u* ∈ *Z* ⟹ ∃ *Z'*. *A* ⊢ ⟨*l*, *Z*⟩ ⇝* ⟨*l'*, *Z'*⟩ ∧ *u'* ∈ *Z'*
⟨*proof*⟩

end
theory *DBM*
imports *Floyd-Warshall Timed-Automata*
begin

3 Difference Bound Matrices

3.1 Definitions

Difference Bound Matrices (DBMs) constrain differences of clocks (or more precisely, the difference of values assigned to individual clocks by a valuation). The possible constraints are given by the following datatype:

datatype ('*t*::*time*) *DBMEntry* = *Le* '*t* | *Lt* '*t* | *INF* (∞)

This yields a simple definition of DBMs:

type-synonym '*t DBM* = *nat* ⇒ *nat* ⇒ '*t DBMEntry*

To relate clocks with rows and columns of a DBM, we use a clock numbering *v* of type '*c* ⇒ *nat* to map clocks to indices. DBMs will regularly be accompanied by a natural number *n*, which designates the number of clocks constrained by the matrix. To be able to represent the full set of clock constraints with DBMs, we add an imaginary clock **0**, which shall be assigned to 0 in every valuation. In the following predicate we explicitly keep track of **0**.

inductive *dbm-entry-val* :: ('*c*, '*t*) *cval* ⇒ '*c option* ⇒ '*c option* ⇒ ('*t*::*time*) *DBMEntry* ⇒ *bool*
where
u r ≤ *d* ⟹ *dbm-entry-val u* (*Some r*) *None* (*Le d*) |
−*u c* ≤ *d* ⟹ *dbm-entry-val u None* (*Some c*) (*Le d*) |
u r < *d* ⟹ *dbm-entry-val u* (*Some r*) *None* (*Lt d*) |
−*u c* < *d* ⟹ *dbm-entry-val u None* (*Some c*) (*Lt d*) |
u r − *u c* ≤ *d* ⟹ *dbm-entry-val u* (*Some r*) (*Some c*) (*Le d*) |
u r − *u c* < *d* ⟹ *dbm-entry-val u* (*Some r*) (*Some c*) (*Lt d*) |
dbm-entry-val - - - ∞

declare *dbm-entry-val.intros*[*intro*]
inductive-cases[*elim*!]: *dbm-entry-val u None* (*Some c*) (*Le d*)
inductive-cases[*elim*!]: *dbm-entry-val u* (*Some c*) *None* (*Le d*)
inductive-cases[*elim*!]: *dbm-entry-val u None* (*Some c*) (*Lt d*)
inductive-cases[*elim*!]: *dbm-entry-val u* (*Some c*) *None* (*Lt d*)
inductive-cases[*elim*!]: *dbm-entry-val u* (*Some r*) (*Some c*) (*Le d*)
inductive-cases[*elim*!]: *dbm-entry-val u* (*Some r*) (*Some c*) (*Lt d*)

fun *dbm-entry-bound* :: ('*t*::*time*) *DBMEntry* ⇒ '*t*
where
dbm-entry-bound (*Le t*) = *t* |
dbm-entry-bound (*Lt t*) = *t* |
dbm-entry-bound ∞ = *0*

inductive *dbm-lt* :: ('*t*::*time*) *DBMEntry* ⇒ '*t DBMEntry* ⇒ *bool*
(- ≺ - [*51*, *51*] *50*)
where
dbm-lt (*Lt* -) ∞ |
dbm-lt (*Le* -) ∞ |
a < *b* ⟹ *dbm-lt* (*Le a*) (*Le b*) |
a < *b* ⟹ *dbm-lt* (*Le a*) (*Lt b*) |
a ≤ *b* ⟹ *dbm-lt* (*Lt a*) (*Le b*) |
a < *b* ⟹ *dbm-lt* (*Lt a*) (*Lt b*)

declare *dbm-lt.intros*[*intro*]

definition *dbm-le* :: ('*t*::*time*) *DBMEntry* ⇒ '*t DBMEntry* ⇒ *bool*
(- ≼ - [*51*, *51*] *50*)
where
dbm-le a b ≡ (*a* ≺ *b*) ∨ *a* = *b*

Now a valuation is contained in the zone represented by a DBM if it fulfills all individual constraints:

definition *DBM-val-bounded* :: $('c \Rightarrow nat) \Rightarrow ('c, 't)\ cval \Rightarrow ('t::time)\ DBM$
$\Rightarrow nat \Rightarrow bool$
where
DBM-val-bounded v u m n $\equiv$ *Le 0* $\preceq$ *m 0 0* $\wedge$
$(\forall\ c.\ v\ c \leq n \longrightarrow (dbm\text{-}entry\text{-}val\ u\ None\ (Some\ c)\ (m\ 0\ (v\ c))$
$\wedge\ dbm\text{-}entry\text{-}val\ u\ (Some\ c)\ None\ (m\ (v\ c)\ 0)))$
$\wedge\ (\forall\ c1\ c2.\ v\ c1 \leq n \wedge v\ c2 \leq n \longrightarrow dbm\text{-}entry\text{-}val\ u\ (Some\ c1)\ (Some\ c2)\ (m\ (v\ c1)\ (v\ c2)))$

abbreviation *DBM-val-bounded-abbrev* ::
$('c, 't)\ cval \Rightarrow ('c \Rightarrow nat) \Rightarrow nat \Rightarrow ('t::time)\ DBM \Rightarrow bool$
$(\text{-} \vdash_{\text{-},\text{-}} \text{-})$
where
$u \vdash_{v,n} M \equiv DBM\text{-}val\text{-}bounded\ v\ u\ M\ n$

abbreviation
dmin a b $\equiv$ *if a* $\prec$ *b then a else b*

lemma *dbm-le-dbm-min*:
$a \preceq b \Longrightarrow a = dmin\ a\ b$ ⟨*proof*⟩

lemma *dbm-lt-asym*:
assumes $e \prec f$
shows $^{\sim} f \prec e$
⟨*proof*⟩

lemma *dbm-le-dbm-min2*:
$a \preceq b \Longrightarrow a = dmin\ b\ a$
⟨*proof*⟩

lemma *dmb-le-dbm-entry-bound-inf*:
$a \preceq b \Longrightarrow a = \infty \Longrightarrow b = \infty$
⟨*proof*⟩

lemma *dbm-not-lt-eq*: $\neg\ a \prec b \Longrightarrow \neg\ b \prec a \Longrightarrow a = b$
⟨*proof*⟩

lemma *dbm-not-lt-impl*: $\neg\ a \prec b \Longrightarrow b \prec a \vee a = b$ ⟨*proof*⟩

lemma *dmin a b* $=$ *dmin b a*
⟨*proof*⟩

lemma *dbm-lt-trans*: $a \prec b \Longrightarrow b \prec c \Longrightarrow a \prec c$
⟨*proof*⟩

lemma *aux-3*: $\neg\ a \prec b \Longrightarrow \neg\ b \prec c \Longrightarrow a \prec c \Longrightarrow c = a$
⟨*proof*⟩

inductive-cases[*elim*!]: $\infty \prec x$

lemma *dbm-lt-asymmetric*[*simp*]: $x \prec y \Longrightarrow y \prec x \Longrightarrow False$
⟨*proof*⟩

lemma *le-dbm-le*: $Le\ a \preceq Le\ b \Longrightarrow a \leq b$ ⟨*proof*⟩

lemma *le-dbm-lt*: $Le\ a \preceq Lt\ b \Longrightarrow a < b$ ⟨*proof*⟩

lemma *lt-dbm-le*: $Lt\ a \preceq Le\ b \Longrightarrow a \leq b$ ⟨*proof*⟩

lemma *lt-dbm-lt*: $Lt\ a \preceq Lt\ b \Longrightarrow a \leq b$ ⟨*proof*⟩

lemma *not-dbm-le-le-impl*: $\neg\ Le\ a \prec Le\ b \Longrightarrow a \geq b$ ⟨*proof*⟩

lemma *not-dbm-lt-le-impl*: $\neg\ Lt\ a \prec Le\ b \Longrightarrow a > b$ ⟨*proof*⟩

lemma *not-dbm-lt-lt-impl*: $\neg\ Lt\ a \prec Lt\ b \Longrightarrow a \geq b$ ⟨*proof*⟩

lemma *not-dbm-le-lt-impl*: $\neg\ Le\ a \prec Lt\ b \Longrightarrow a \geq b$ ⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proo*
theory *Paths-Cycles*
imports *Floyd-Warshall Timed-Automata*
begin

4 Library for Paths, Arcs and Lengths

lemma *length-eq-distinct*:
assumes *set xs = set ys distinct xs length xs = length ys*
shows *distinct ys*
⟨*proof*⟩

4.1 Arcs

fun *arcs* :: *nat* ⇒ *nat* ⇒ *nat list* ⇒ (*nat* ∗ *nat*) *list* **where**
arcs a b [] = [(*a*,*b*)] |
arcs a b (*x* # *xs*) = (*a*, *x*) # *arcs x b xs*

definition *arcs′* :: *nat list* ⇒ (*nat* ∗ *nat*) *set* **where**
arcs′ xs = *set* (*arcs* (*hd xs*) (*last xs*) (*butlast* (*tl xs*)))

lemma *arcs′-decomp*:
length xs > *1* ⟹ (*i*, *j*) ∈ *arcs′ xs* ⟹ ∃ *zs ys*. *xs* = *zs* @ *i* # *j* # *ys*
⟨*proof*⟩

lemma *arcs-decomp-tail*:
arcs j l (*ys* @ [*i*]) = *arcs j i ys* @ [(*i*, *l*)]
⟨*proof*⟩

lemma *arcs-decomp*: *xs* = *ys* @ *y* # *zs* ⟹ *arcs x z xs* = *arcs x y ys* @ *arcs y z zs*
⟨*proof*⟩

lemma *distinct-arcs-ex*:
distinct xs ⟹ *i* ∉ *set xs* ⟹ *xs* ≠ [] ⟹ ∃ *a b*. *a* ≠ *x* ∧ (*a*,*b*) ∈ *set* (*arcs i j xs*)
⟨*proof*⟩

lemma *cycle-rotate-2-aux*:
(*i*, *j*) ∈ *set* (*arcs a b* (*xs* @ [*c*])) ⟹ (*i*,*j*) ≠ (*c*,*b*) ⟹ (*i*, *j*) ∈ *set* (*arcs a c xs*)
⟨*proof*⟩

lemma *arcs-set-elem1*:
assumes *j* ≠ *k k* ∈ *set* (*i* # *xs*)
shows ∃ *l*. (*k*, *l*) ∈ *set* (*arcs i j xs*) ⟨*proof*⟩

lemma *arcs-set-elem2*:
assumes *i* ≠ *k k* ∈ *set* (*j* # *xs*)
shows ∃ *l*. (*l*, *k*) ∈ *set* (*arcs i j xs*) ⟨*proof*⟩

4.2 Length of Paths

lemmas (**in** *linordered-ab-monoid-add*) *comm* = *add.commute*

lemma *len-add*:
fixes *M* :: (′*a* :: *linordered-ab-monoid-add*) *mat*
shows *len M i j xs* + *len M i j xs* = *len* (λ*i j*. *M i j* + *M i j*) *i j xs*
⟨*proof*⟩

4.3 Canonical Matrices

abbreviation
canonical M n ≡ ∀ *i j k*. *i* ≤ *n* ∧ *j* ≤ *n* ∧ *k* ≤ *n* ⟶ *M i k* ≤ *M i j* + *M j k*

lemma *fw-canonical*:
cycle-free m n $\Longrightarrow$ *canonical* (*fw m n n n n*) *n*
⟨*proof*⟩

lemma *canonical-len*:
canonical M n $\Longrightarrow$ $i \leq n \Longrightarrow j \leq n \Longrightarrow$ *set xs* $\subseteq \{0..n\} \Longrightarrow$ *M i j* $\leq$ *len M i j xs*
⟨*proof*⟩

4.4 Cycle Rotation

lemma *cycle-rotate*:
fixes *M* :: (′*a* :: *linordered-ab-monoid-add*) *mat*
assumes *length xs* > *1* (*i*, *j*) ∈ *arcs*′ *xs*
shows ∃ *ys zs*. *len M a a xs* = *len M i i* (*j* # *ys* @ *a* # *zs*) ∧ *xs* = *zs* @ *i* # *j* # *ys* ⟨*proof*⟩

lemma *cycle-rotate-2*:
fixes *M* :: (′*a* :: *linordered-ab-monoid-add*) *mat*
assumes *xs* ≠ [] (*i*, *j*) ∈ *set* (*arcs a a xs*)
shows ∃ *ys*. *len M a a xs* = *len M i i* (*j* # *ys*) ∧ *set ys* ⊆ *set* (*a* # *xs*) ∧ *length ys* < *length xs*
⟨*proof*⟩

lemma *cycle-rotate-len-arcs*:
fixes *M* :: (′*a* :: *linordered-ab-monoid-add*) *mat*
assumes *length xs* > *1* (*i*, *j*) ∈ *arcs*′ *xs*
shows ∃ *ys zs*. *len M a a xs* = *len M i i* (*j* # *ys* @ *a* # *zs*)
∧ *set* (*arcs a a xs*) = *set* (*arcs i i* (*j* # *ys* @ *a* # *zs*)) ∧ *xs* = *zs* @ *i* # *j* # *ys*
⟨*proof*⟩

lemma *cycle-rotate-2*′:
fixes *M* :: (′*a* :: *linordered-ab-monoid-add*) *mat*
assumes *xs* ≠ [] (*i*, *j*) ∈ *set* (*arcs a a xs*)
shows ∃ *ys*. *len M a a xs* = *len M i i* (*j* # *ys*) ∧ *set* (*i* # *j* # *ys*) = *set* (*a* # *xs*)
∧ *1* + *length ys* = *length xs* ∧ *set* (*arcs a a xs*) = *set* (*arcs i i* (*j* # *ys*))
⟨*proof*⟩

4.5 Equivalent Characterizations of Cycle-Freeness

lemma *negative-cycle-dest-diag*:
$\neg$ *cycle-free M n* $\Longrightarrow \exists$ *i xs. i* $\leq$ *n* $\wedge$ *set xs* $\subseteq \{0..n\}$ $\wedge$ *len M i i xs* $<$ **1**
⟨*proof*⟩

abbreviation *cyc-free* :: (*'a*::*linordered-ab-monoid-add*) *mat* $\Rightarrow$ *nat* $\Rightarrow$ *bool*
where
cyc-free m n $\equiv \forall$ *i xs. i* $\leq$ *n* $\wedge$ *set xs* $\subseteq \{0..n\} \longrightarrow$ *len m i i xs* $\geq$ **1**

lemma *cycle-free-diag-intro*:
cyc-free M n $\Longrightarrow$ *cycle-free M n*
⟨*proof*⟩

lemma *cycle-free-diag-equiv*:
cyc-free M n $\longleftrightarrow$ *cycle-free M n* ⟨*proof*⟩

lemma *cycle-free-diag-dest*:
cycle-free M n $\Longrightarrow$ *cyc-free M n*
⟨*proof*⟩

lemma *cyc-free-diag-dest*:
assumes *cyc-free M n i* $\leq$ *n set xs* $\subseteq \{0..n\}$
shows *len M i i xs* $\geq$ **1**
⟨*proof*⟩

lemma *cycle-free-0-0*:
fixes *M* :: (*'a*::*linordered-ab-monoid-add*) *mat*
assumes *cycle-free M n*
shows *M 0 0* $\geq$ **1**
⟨*proof*⟩

4.6 More Theorems Related to Floyd-Warshall

lemma *D-cycle-free-len-dest*:
cycle-free m n
$\Longrightarrow \forall$ *i* $\leq$ *n.* $\forall$ *j* $\leq$ *n. D m i j n* = *m' i j* $\Longrightarrow$ *i* $\leq$ *n* $\Longrightarrow$ *j* $\leq$ *n* $\Longrightarrow$ *set xs* $\subseteq \{0..n\}$
$\Longrightarrow \exists$ *ys. set ys* $\subseteq \{0..n\}$ $\wedge$ *len m' i j xs* = *len m i j ys*
⟨*proof*⟩

lemma *D-cyc-free-preservation*:
cyc-free m n $\Longrightarrow \forall$ *i* $\leq$ *n.* $\forall$ *j* $\leq$ *n. D m i j n* = *m' i j* $\Longrightarrow$ *cyc-free m' n*
⟨*proof*⟩

abbreviation $FW\ m\ n \equiv fw\ m\ n\ n\ n\ n$

lemma *FW-cyc-free-preservation*:
$cyc\text{-}free\ m\ n \Longrightarrow cyc\text{-}free\ (FW\ m\ n)\ n$
⟨*proof*⟩

lemma *cyc-free-diag-dest'*:
$cyc\text{-}free\ m\ n \Longrightarrow i \leq n \Longrightarrow m\ i\ i \geq \mathbf{1}$
⟨*proof*⟩

lemma *FW-diag-neutral-preservation*:
$\forall\ i \leq n.\ M\ i\ i = \mathbf{1} \Longrightarrow cyc\text{-}free\ M\ n \Longrightarrow \forall\ i{\leq}n.\ (FW\ M\ n)\ i\ i = \mathbf{1}$
⟨*proof*⟩

lemma *FW-fixed-preservation*:
fixes $M :: ('a{::}linordered\text{-}ab\text{-}monoid\text{-}add)\ mat$
assumes A: $i \leq n\ M\ 0\ i + M\ i\ 0 = \mathbf{1}\ canonical\ (FW\ M\ n)\ n\ cyc\text{-}free\ (FW\ M\ n)\ n$
shows $FW\ M\ n\ 0\ i + FW\ M\ n\ i\ 0 = \mathbf{1}$ ⟨*proof*⟩

lemma *diag-cyc-free-neutral*:
$cyc\text{-}free\ M\ n \Longrightarrow \forall\, k{\leq}n.\ M\ k\ k \leq \mathbf{1} \Longrightarrow \forall\, i{\leq}n.\ M\ i\ i = \mathbf{1}$
⟨*proof*⟩

lemma *fw-upd-canonical-id*:
$canonical\ M\ n \Longrightarrow i \leq n \Longrightarrow j \leq n \Longrightarrow k \leq n \Longrightarrow fw\text{-}upd\ M\ k\ i\ j = M$
⟨*proof*⟩

lemma *fw-canonical-id*:
$canonical\ M\ n \Longrightarrow i \leq n \Longrightarrow j \leq n \Longrightarrow k \leq n \Longrightarrow fw\ M\ n\ k\ i\ j = M$
⟨*proof*⟩

lemmas *FW-canonical-id* = *fw-canonical-id*[*OF - order.refl order.refl order.refl*]

4.7 Helper Lemmas for Bouyer's Theorem on Approximation

lemma *aux1*: $i \leq n \Longrightarrow j \leq n \Longrightarrow set\ xs \subseteq \{0..n\} \Longrightarrow (a,b) \in set\ (arcs\ i\ j\ xs) \Longrightarrow a \leq n \wedge b \leq n$
⟨*proof*⟩

lemma *arcs-distinct1*:

$i \notin set\ xs \Longrightarrow j \notin set\ xs \Longrightarrow distinct\ xs \Longrightarrow xs \neq [] \Longrightarrow (a,b) \in set\ (arcs\ i\ j\ xs) \Longrightarrow a \neq b$
⟨*proof*⟩

lemma *arcs-distinct2*:
$i \notin set\ xs \Longrightarrow j \notin set\ xs \Longrightarrow distinct\ xs \Longrightarrow i \neq j \Longrightarrow (a,b) \in set\ (arcs\ i\ j\ xs) \Longrightarrow a \neq b$
⟨*proof*⟩

lemma *arcs-distinct3*: $distinct\ (a\ \#\ b\ \#\ c\ \#\ xs) \Longrightarrow (i,j) \in set\ (arcs\ a\ b\ xs) \Longrightarrow i \neq c \wedge j \neq c$
⟨*proof*⟩

lemma *arcs-elem*:
assumes $(a,\ b) \in set\ (arcs\ i\ j\ xs)$ **shows** $a \in set\ (i\ \#\ xs)\ b \in set\ (j\ \#\ xs)$
⟨*proof*⟩

lemma *arcs-distinct-dest1*:
$distinct\ (i\ \#\ a\ \#\ zs) \Longrightarrow (b,c) \in set\ (arcs\ a\ j\ zs) \Longrightarrow b \neq i$
⟨*proof*⟩

lemma *arcs-distinct-fix*:
$distinct\ (a\ \#\ x\ \#\ xs\ @\ [b]) \Longrightarrow (a,c) \in set\ (arcs\ a\ b\ (x\ \#\ xs)) \Longrightarrow c = x$
⟨*proof*⟩

lemma *disjE3*: $A \vee B \vee C \Longrightarrow (A \Longrightarrow G) \Longrightarrow (B \Longrightarrow G) \Longrightarrow (C \Longrightarrow G) \Longrightarrow G$
⟨*proof*⟩

lemma *arcs-predecessor*:
assumes $(a,\ b) \in set\ (arcs\ i\ j\ xs)\ a \neq i$
shows $\exists\ c.\ (c,\ a) \in set\ (arcs\ i\ j\ xs)$ ⟨*proof*⟩

lemma *arcs-successor*:
assumes $(a,\ b) \in set\ (arcs\ i\ j\ xs)\ b \neq j$
shows $\exists\ c.\ (b,c) \in set\ (arcs\ i\ j\ xs)$ ⟨*proof*⟩

lemma *arcs-predecessor′*:
assumes $(a,\ b) \in set\ (arcs\ i\ j\ (x\ \#\ xs))\ (a,b) \neq (i,\ x)$
shows $\exists\ c.\ (c,\ a) \in set\ (arcs\ i\ j\ (x\ \#\ xs))$ ⟨*proof*⟩

lemma *arcs-cases*:
assumes $(a,\ b) \in set\ (arcs\ i\ j\ xs)\ xs \neq []$

shows $(\exists\ ys.\ xs = b \# ys \wedge a = i) \vee (\exists\ ys.\ xs = ys\ @\ [a] \wedge b = j)$
$\vee\ (\exists\ c\ d\ ys.\ (a,b) \in set\ (arcs\ c\ d\ ys) \wedge xs = c \# ys\ @\ [d])$
⟨*proof*⟩

lemma *arcs-cases'*:
assumes $(a,\ b) \in set\ (arcs\ i\ j\ xs)\ xs \neq []$
shows $(\exists\ ys.\ xs = b \# ys \wedge a = i) \vee (\exists\ ys.\ xs = ys\ @\ [a] \wedge b = j) \vee (\exists\ ys\ zs.\ xs = ys\ @\ a \# b \# zs)$
⟨*proof*⟩

lemma *arcs-successor'*:
assumes $(a,\ b) \in set\ (arcs\ i\ j\ xs)\ b \neq j$
shows $\exists\ c.\ xs = [b] \wedge a = i \vee (\exists\ ys.\ xs = b \# c \# ys \wedge a = i) \vee (\exists\ ys.\ xs = ys\ @\ [a,b] \wedge c = j)$
$\vee\ (\exists\ ys\ zs.\ xs = ys\ @\ a \# b \# c \# zs)$
⟨*proof*⟩

lemma *list-last*:
$xs = [] \vee (\exists\ y\ ys.\ xs = ys\ @\ [y])$
⟨*proof*⟩

lemma *arcs-predecessor''*:
assumes $(a,\ b) \in set\ (arcs\ i\ j\ xs)\ a \neq i$
shows $\exists\ c.\ xs = [a] \vee (\exists\ ys.\ xs = a \# b \# ys) \vee (\exists\ ys.\ xs = ys\ @\ [c,a] \wedge b = j)$
$\vee\ (\exists\ ys\ zs.\ xs = ys\ @\ c \# a \# b \# zs)$
⟨*proof*⟩

lemma *arcs-ex-middle*:
$\exists\ b.\ (a,\ b) \in set\ (arcs\ i\ j\ (ys\ @\ a \# xs))$
⟨*proof*⟩

lemma *arcs-ex-head*:
$\exists\ b.\ (i,\ b) \in set\ (arcs\ i\ j\ xs)$
⟨*proof*⟩

4.7.1 Successive

fun *successive* **where**
successive - [] = True |
successive P [-] = True |
successive P (x # y # xs) ⟷ ¬ P y ∧ successive P xs ∨ ¬ P x ∧ successive P (y # xs)

lemma ¬ *successive* (λ *x*. *x* > (*0* :: *nat*)) [*Suc 0*, *Suc 0*] ⟨*proof*⟩
lemma *successive* (λ *x*. *x* > (*0* :: *nat*)) [*Suc 0*] ⟨*proof*⟩
lemma *successive* (λ *x*. *x* > (*0* :: *nat*)) [*Suc 0*, *0*, *Suc 0*] ⟨*proof*⟩
lemma ¬ *successive* (λ *x*. *x* > (*0* :: *nat*)) [*Suc 0*, *0*, *Suc 0*, *Suc 0*] ⟨*proof*⟩
lemma ¬ *successive* (λ *x*. *x* > (*0* :: *nat*)) [*Suc 0*, *0*, *0*, *Suc 0*, *Suc 0*] ⟨*proof*⟩
lemma *successive* (λ *x*. *x* > (*0* :: *nat*)) [*Suc 0*, *0*, *Suc 0*, *0*, *Suc 0*] ⟨*proof*⟩
lemma ¬ *successive* (λ *x*. *x* > (*0* :: *nat*)) [*Suc 0*, *Suc 0*, *0*, *Suc 0*] ⟨*proof*⟩
lemma *successive* (λ *x*. *x* > (*0* :: *nat*)) [*0*, *0*, *Suc 0*, *0*] ⟨*proof*⟩

lemma *successive-step*: *successive P* (*x* # *xs*) ⟹ ¬ *P x* ⟹ *successive P xs*
⟨*proof*⟩

lemma *successive-step-2*: *successive P* (*x* # *y* # *xs*) ⟹ ¬ *P y* ⟹ *successive P xs*
⟨*proof*⟩

lemma *successive-stepI*:
successive P xs ⟹ ¬ *P x* ⟹ *successive P* (*x* # *xs*)
⟨*proof*⟩

theorem *list-two-induct*[*case-names Nil Single Cons*]:
fixes *P* :: *'a list* ⇒ *bool*
and *list* :: *'a list*
assumes *Nil*: *P* []
assumes *Single*: ⋀ *x*. *P* [*x*]
and *Cons*: ⋀*x1 x2 xs*. *P xs* ⟹ *P* (*x2* # *xs*) ⟹ *P* (*x1* # *x2* # *xs*)
shows *P xs*
⟨*proof*⟩

lemma *successive-end-1*:
successive P xs ⟹ ¬ *P x* ⟹ *successive P* (*xs* @ [*x*])
⟨*proof*⟩

lemma *successive-ends-1*:
successive P xs ⟹ ¬ *P x* ⟹ *successive P ys* ⟹ *successive P* (*xs* @ *x* # *ys*)
⟨*proof*⟩

lemma *successive-ends-1'*:
successive P xs ⟹ ¬ *P x* ⟹ *P y* ⟹ ¬ *P z* ⟹ *successive P ys* ⟹ *successive P* (*xs* @ *x* # *y* # *z* # *ys*)
⟨*proof*⟩

lemma *successive-end-2*:
successive P xs $\Longrightarrow$ $\neg$ *P x* $\Longrightarrow$ *successive P* (*xs* @ [*x*,*y*])
⟨*proof*⟩

lemma *successive-end-2′*:
successive P (*xs* @ [*x*]) $\Longrightarrow$ $\neg$ *P x* $\Longrightarrow$ *successive P* (*xs* @ [*x*,*y*])
⟨*proof*⟩

lemma *successive-end-3*:
successive P (*xs* @ [*x*]) $\Longrightarrow$ $\neg$ *P x* $\Longrightarrow$ *P y* $\Longrightarrow$ $\neg$ *P z* $\Longrightarrow$ *successive P* (*xs* @ [*x*,*y*,*z*])
⟨*proof*⟩

lemma *successive-step-rev*:
successive P (*xs* @ [*x*]) $\Longrightarrow$ $\neg$ *P x* $\Longrightarrow$ *successive P xs*
⟨*proof*⟩

lemma *successive-glue*:
successive P (*zs* @ [*z*]) $\Longrightarrow$ *successive P* (*x* # *xs*) $\Longrightarrow$ $\neg$ *P z* $\vee$ $\neg$ *P x* $\Longrightarrow$ *successive P* (*zs* @ [*z*] @ *x* # *xs*)
⟨*proof*⟩

lemma *successive-glue′*:
successive P (*zs* @ [*y*]) $\wedge$ $\neg$ *P z* $\vee$ *successive P zs* $\wedge$ $\neg$ *P y*
$\Longrightarrow$ *successive P* (*x* # *xs*) $\wedge$ $\neg$ *P w* $\vee$ *successive P xs* $\wedge$ $\neg$ *P x*
$\Longrightarrow$ $\neg$ *P z* $\vee$ $\neg$ *P w* $\Longrightarrow$ *successive P* (*zs* @ *y* # *z* # *w* # *x* # *xs*)
⟨*proof*⟩

lemma *successive-dest-head*:
xs = *w* # *x* # *ys* $\Longrightarrow$ *successive P xs* $\Longrightarrow$ *successive P* (*x* # *xs*) $\wedge$ $\neg$ *P w* $\vee$ *successive P xs* $\wedge$ $\neg$ *P x*
⟨*proof*⟩

lemma *successive-dest-tail*:
xs = *zs* @ [*y*,*z*] $\Longrightarrow$ *successive P xs* $\Longrightarrow$ *successive P* (*zs* @ [*y*]) $\wedge$ $\neg$ *P z* $\vee$ *successive P zs* $\wedge$ $\neg$ *P y*
⟨*proof*⟩

lemma *successive-split*:
xs = *ys* @ *zs* $\Longrightarrow$ *successive P xs* $\Longrightarrow$ *successive P ys* $\wedge$ *successive P zs*
⟨*proof*⟩

lemma *successive-decomp*:

$xs = x \# ys \mathbin{@} zs \mathbin{@} [z] \Longrightarrow successive\ P\ xs \Longrightarrow \neg\ P\ x \vee \neg\ P\ z \Longrightarrow successive\ P\ (zs \mathbin{@} [z] \mathbin{@} (x \# ys))$
⟨*proof*⟩

lemma *successive-predecessor*:
assumes $(a,\ b) \in set\ (arcs\ i\ j\ xs)\ a \neq i\ successive\ P\ (arcs\ i\ j\ xs)\ P\ (a,b)\ xs \neq []$
shows $\exists\ c.\ (xs = [a] \wedge c = i \vee (\exists\ ys.\ xs = a \# b \# ys \wedge c = i) \vee (\exists\ ys.\ xs = ys \mathbin{@} [c,a] \wedge b = j)$
$\vee\ (\exists\ ys\ zs.\ xs = ys \mathbin{@} c \# a \# b \# zs)) \wedge \neg\ P\ (c,a)$
⟨*proof*⟩

thm *arcs-successor′*

lemma *successive-successor*:
assumes $(a,\ b) \in set\ (arcs\ i\ j\ xs)\ b \neq j\ successive\ P\ (arcs\ i\ j\ xs)\ P\ (a,b)\ xs \neq []$
shows $\exists\ c.\ (xs = [b] \wedge c = j \vee (\exists\ ys.\ xs = b \# c \# ys) \vee (\exists\ ys.\ xs = ys \mathbin{@} [a,b] \wedge c = j)$
$\vee\ (\exists\ ys\ zs.\ xs = ys \mathbin{@} a \# b \# c \# zs)) \wedge \neg\ P\ (b,c)$
⟨*proof*⟩

lemmas *add-mono-right* = *add-mono*[*OF order-refl*]
lemmas *add-mono-left* = *add-mono*[*OF - order-refl*]

Obtaining successive and distinct paths **lemma** *canonical-successive*:
fixes $A\ B$
defines $M \equiv \lambda\ i\ j.\ min\ (A\ i\ j)\ (B\ i\ j)$
assumes *canonical* $A\ n$
assumes $set\ xs \subseteq \{0..n\}$
assumes $i \leq n\ j \leq n$
shows $\exists\ ys.\ len\ M\ i\ j\ ys \leq len\ M\ i\ j\ xs \wedge set\ ys \subseteq \{0..n\}$
$\wedge\ successive\ (\lambda\ (a,\ b).\ M\ a\ b = A\ a\ b)\ (arcs\ i\ j\ ys)$
⟨*proof*⟩

lemma *canonical-successive-distinct*:
fixes $A\ B$
defines $M \equiv \lambda\ i\ j.\ min\ (A\ i\ j)\ (B\ i\ j)$
assumes *canonical* $A\ n$
assumes $set\ xs \subseteq \{0..n\}$
assumes $i \leq n\ j \leq n$
assumes $distinct\ xs\ i \notin set\ xs\ j \notin set\ xs$
shows $\exists\ ys.\ len\ M\ i\ j\ ys \leq len\ M\ i\ j\ xs \wedge set\ ys \subseteq set\ xs$

$\wedge$ *successive* (λ (*a*, *b*). *M a b* = *A a b*) (*arcs i j ys*)
$\wedge$ *distinct ys* $\wedge$ *i* $\notin$ *set ys* $\wedge$ *j* $\notin$ *set ys*
⟨*proof*⟩

lemma *successive-snd-last*: *successive P* (*xs* @ [*x*, *y*]) $\Longrightarrow$ *P y* $\Longrightarrow$ $\neg$ *P x*
⟨*proof*⟩

lemma *canonical-shorten-rotate-neg-cycle*:
fixes *A B*
defines *M* $\equiv$ λ *i j*. *min* (*A i j*) (*B i j*)
assumes *canonical A n*
assumes *set xs* $\subseteq$ {*0*..*n*}
assumes *i* $\leq$ *n*
assumes *len M i i xs* < **1**
shows $\exists$ *j ys*. *len M j j ys* < **1** $\wedge$ *set* (*j* # *ys*) $\subseteq$ *set* (*i* # *xs*)
$\wedge$ *successive* (λ (*a*, *b*). *M a b* = *A a b*) (*arcs j j ys*)
$\wedge$ *distinct ys* $\wedge$ *j* $\notin$ *set ys* $\wedge$
(*ys* $\neq$ [] $\longrightarrow$ *M j* (*hd ys*) $\neq$ *A j* (*hd ys*) $\vee$ *M* (*last ys*) *j* $\neq$ *A* (*last ys*) *j*)
⟨*proof*⟩

lemma *successive-arcs-extend-last*:
successive P (*arcs i j xs*) $\Longrightarrow$ $\neg$ *P* (*i*, *hd xs*) $\vee$ $\neg$ *P* (*last xs*, *j*) $\Longrightarrow$ *xs* $\neq$ []
$\Longrightarrow$ *successive P* (*arcs i j xs* @ [(*i*, *hd xs*)])
⟨*proof*⟩

lemma *cycle-rotate-arcs*:
fixes *M* :: ('*a* :: *linordered-ab-monoid-add*) *mat*
assumes *length xs* > *1* (*i*, *j*) $\in$ *arcs*′ *xs*
shows $\exists$ *ys zs*. *set* (*arcs a a xs*) = *set* (*arcs i i* (*j* # *ys* @ *a* # *zs*)) $\wedge$ *xs* = *zs* @ *i* # *j* # *ys* ⟨*proof*⟩

lemma *cycle-rotate-len-arcs-successive*:
fixes *M* :: ('*a* :: *linordered-ab-monoid-add*) *mat*
assumes *length xs* > *1* (*i*, *j*) $\in$ *arcs*′ *xs successive P* (*arcs a a xs*) $\neg$ *P* (*a*, *hd xs*) $\vee$ $\neg$ *P* (*last xs*, *a*)
shows $\exists$ *ys zs*. *len M a a xs* = *len M i i* (*j* # *ys* @ *a* # *zs*)
$\wedge$ *set* (*arcs a a xs*) = *set* (*arcs i i* (*j* # *ys* @ *a* # *zs*)) $\wedge$ *xs* = *zs* @ *i* # *j* # *ys*
$\wedge$ *successive P* (*arcs i i* (*j* # *ys* @ *a* # *zs*))
⟨*proof*⟩

lemma *successive-successors*:

$xs = ys$ @ a # b # c # $zs \Longrightarrow$ *successive* P (*arcs* i j xs) $\Longrightarrow \neg$ P (a,b) $\vee \neg$ P (b, c)
⟨*proof*⟩

lemma *successive-successors'*:
$xs = ys$ @ a # b # $zs \Longrightarrow$ *successive* P $xs \Longrightarrow \neg$ P $a \vee \neg$ P b
⟨*proof*⟩

lemma *cycle-rotate-len-arcs-successive'*:
fixes M :: ('a :: *linordered-ab-monoid-add*) *mat*
assumes *length* $xs > 1$ (i, j) $\in$ *arcs'* xs *successive* P (*arcs* a a xs) $\neg$ P (a, *hd* xs) $\vee \neg$ P (*last* xs, a)
shows $\exists$ ys zs. *len* M a a xs = *len* M i i (j # ys @ a # zs) $\wedge$ *set* (*arcs* a a xs) = *set* (*arcs* i i (j # ys @ a # zs)) $\wedge$ xs = zs @ i # j # ys $\wedge$ *successive* P (*arcs* i i (j # ys @ a # zs) @ [(i,j)])
⟨*proof*⟩

lemma *cycle-rotate-3*:
fixes M :: ('a :: *linordered-ab-monoid-add*) *mat*
assumes $xs \neq$ [] (i, j) $\in$ *set* (*arcs* a a xs) *successive* P (*arcs* a a xs) $\neg$ P (a, *hd* xs) $\vee \neg$ P (*last* xs, a)
shows $\exists$ ys. *len* M a a xs = *len* M i i (j # ys) $\wedge$ *set* (i # j # ys) = *set* (a # xs) $\wedge$ 1 + *length* ys = *length* xs $\wedge$ *set* (*arcs* a a xs) = *set* (*arcs* i i (j # ys)) $\wedge$ *successive* P (*arcs* i i (j # ys))
⟨*proof*⟩

lemma *cycle-rotate-3'*:
fixes M :: ('a :: *linordered-ab-monoid-add*) *mat*
assumes $xs \neq$ [] (i, j) $\in$ *set* (*arcs* a a xs) *successive* P (*arcs* a a xs) $\neg$ P (a, *hd* xs) $\vee \neg$ P (*last* xs, a)
shows $\exists$ ys. *len* M a a xs = *len* M i i (j # ys) $\wedge$ *set* (i # j # ys) = *set* (a # xs) $\wedge$ 1 + *length* ys = *length* xs $\wedge$ *set* (*arcs* a a xs) = *set* (*arcs* i i (j # ys)) $\wedge$ *successive* P (*arcs* i i (j # ys) @ [(i, j)])
⟨*proof*⟩

end
theory *DBM-Basics*
imports *DBM Paths-Cycles*
begin

fun *get-const* **where**

get-const (*Le c*) = *c* |
get-const (*Lt c*) = *c* |
get-const ∞ = *undefined*

4.7.2 Discourse on updating DBMs

abbreviation *DBM-update* :: ('*t*::*time*) *DBM* ⇒ *nat* ⇒ *nat* ⇒ ('*t DBMEntry*) ⇒ ('*t*::*time*) *DBM*
where
DBM-update M m n v ≡ (λ *x y*. *if m* = *x* ∧ *n* = *y then v else M x y*)

fun *DBM-upd* :: ('*t*::*time*) *DBM* ⇒ (*nat* ⇒ *nat* ⇒ '*t DBMEntry*) ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ '*t DBM*
where
DBM-upd M f 0 0 - = DBM-update M 0 0 (*f 0 0*) |
DBM-upd M f (*Suc i*) *0 n = DBM-update* (*DBM-upd M f i n n*) (*Suc i*) *0* (*f* (*Suc i*) *0*) |
DBM-upd M f i (*Suc j*) *n = DBM-update* (*DBM-upd M f i j n*) *i* (*Suc j*) (*f i* (*Suc j*))

lemma *upd-1*:
assumes *j* ≤ *n*
shows *DBM-upd M1 f* (*Suc m*) *n N* (*Suc m*) *j = DBM-upd M1 f* (*Suc m*) *j N* (*Suc m*) *j*
⟨*proof*⟩

lemma *upd-2*:
assumes *i* ≤ *m*
shows *DBM-upd M1 f* (*Suc m*) *n N i j = DBM-upd M1 f* (*Suc m*) *0 N i j*
⟨*proof*⟩

lemma *upd-3*:
assumes *m* ≤ *N n* ≤ *N j* ≤ *n i* ≤ *m*
shows (*DBM-upd M1 f m n N*) *i j* = (*DBM-upd M1 f i j N*) *i j*
⟨*proof*⟩

lemma *upd-id*:
assumes *m* ≤ *N n* ≤ *N i* ≤ *m j* ≤ *n*
shows (*DBM-upd M1 f m n N*) *i j* = *f i j*
⟨*proof*⟩

4.7.3 Zones and DBMs

definition *DBM-zone-repr* :: ('*t*::*time*) *DBM* ⇒ ('*c* ⇒ *nat*) ⇒ *nat* ⇒ ('*c*, '*t* :: *time*) *zone*
([-]-,- [*72*,*72*,*72*] *72*)
where
$[M]_{v,n} = \{u$. *DBM-val-bounded v u M n*$\}$

lemma *dbm-entry-val-mono-1*:
dbm-entry-val u (*Some c*) (*Some c'*) *b* ⟹ $b \preceq b'$ ⟹ *dbm-entry-val u* (*Some c*) (*Some c'*) *b'*
⟨*proof*⟩

lemma *dbm-entry-val-mono-2*:
dbm-entry-val u None (*Some c*) *b* ⟹ $b \preceq b'$ ⟹ *dbm-entry-val u None* (*Some c*) *b'*
⟨*proof*⟩

lemma *dbm-entry-val-mono-3*:
dbm-entry-val u (*Some c*) *None b* ⟹ $b \preceq b'$ ⟹ *dbm-entry-val u* (*Some c*) *None b'*
⟨*proof*⟩

lemma *DBM-le-subset*:
$\forall\ i\ j.\ i \leq n \longrightarrow j \leq n \longrightarrow M\ i\ j \preceq M'\ i\ j \Longrightarrow u \in [M]_{v,n} \Longrightarrow u \in [M']_{v,n}$
⟨*proof*⟩

4.7.4 DBMs Without Negative Cycles are Non-Empty

We need all of these assumptions for the proof that matrices without negative cycles represent non-negative zones: * Abelian (linearly ordered) monoid * Time is non-trivial * Time is dense

lemmas (**in** *linordered-ab-monoid-add*) *comm* = *add.commute*

lemma *sum-gt-neutral-dest'*:
$(a ::$ (('*a* :: *time*) *DBMEntry*)) $\geq \mathbf{1} \Longrightarrow a + b > \mathbf{1} \Longrightarrow \exists\ d.\ Le\ d \leq a \wedge Le\ (-d) \leq b \wedge d \geq 0$
⟨*proof*⟩

lemma *sum-gt-neutral-dest*:
$(a ::$ (('*a* :: *time*) *DBMEntry*)) $+ b > \mathbf{1} \Longrightarrow \exists\ d.\ Le\ d \leq a \wedge Le\ (-d) \leq b$
⟨*proof*⟩

4.7.5 Negative Cycles in DBMs

lemma *DBM-val-bounded-neg-cycle1*:
fixes *i xs* **assumes**
bounded: *DBM-val-bounded v u M n* **and** *A*:$i \leq n$ *set xs* $\subseteq \{0..n\}$ *len M i i xs* $<$ **1 and**
surj-on: $\forall\ k \leq n.\ k > 0 \longrightarrow (\exists\ c.\ v\ c = k)$ **and** *at-most*: $i \neq 0$ *cnt 0 xs* ≤ 1
shows *False*
⟨*proof*⟩

lemma *cnt-0-I*:
$x \notin$ *set xs* $\Longrightarrow$ *cnt x xs* $= 0$
⟨*proof*⟩

lemma *distinct-cnt*: *distinct xs* $\Longrightarrow$ *cnt x xs* ≤ 1
⟨*proof*⟩

lemma *DBM-val-bounded-neg-cycle*:
fixes *i xs* **assumes**
bounded: *DBM-val-bounded v u M n* **and** *A*:$i \leq n$ *set xs* $\subseteq \{0..n\}$ *len M i i xs* $<$ **1 and**
surj-on: $\forall\ k \leq n.\ k > 0 \longrightarrow (\exists\ c.\ v\ c = k)$
shows *False*
⟨*proof*⟩

4.7.6 Floyd-Warshall Algorithm Preservers Zones

lemma *D-dest*: $x = D\ m\ i\ j\ k \Longrightarrow$
$x \in \{len\ m\ i\ j\ xs\ |xs.\ set\ xs \subseteq \{0..k\} \wedge i \notin set\ xs \wedge j \notin set\ xs \wedge distinct\ xs\}$
⟨*proof*⟩

lemma *FW-zone-equiv*:
$\forall\ k \leq n.\ k > 0 \longrightarrow (\exists\ c.\ v\ c = k) \Longrightarrow [M]_{v,n} = [FW\ M\ n]_{v,n}$
⟨*proof*⟩

lemma *new-negative-cycle-aux′*:
fixes $M :: ('a :: time)\ DBM$
fixes *i j d*
defines $M' \equiv \lambda\ i'\ j'.$ *if* $(i' = i \wedge j' = j)$ *then Le d*
else if $(i' = j \wedge j' = i)$ *then Le* $(-d)$
else M i′ j′
assumes $i \leq n\ j \leq n$ *set xs* $\subseteq \{0..n\}$ *cycle-free M n length xs* $= m$

assumes *len M′ i i (j # xs) < 1 ∨ len M′ j j (i # xs) < 1*
assumes *i ≠ j*
shows *∃ xs. set xs ⊆ {0..n} ∧ j ∉ set xs ∧ i ∉ set xs*
∧ (len M′ i i (j # xs) < 1 ∨ len M′ j j (i # xs) < 1) ⟨proof⟩

lemma *new-negative-cycle-aux*:
fixes *M :: ('a :: time) DBM*
fixes *i d*
defines *M′ ≡ λ i′ j′. if (i′ = i ∧ j′ = 0) then Le d*
else if (i′ = 0 ∧ j′ = i) then Le (−d)
else M i′ j′
assumes *i ≤ n set xs ⊆ {0..n} cycle-free M n length xs = m*
assumes *len M′ 0 0 (i # xs) < 1 ∨ len M′ i i (0 # xs) < 1*
assumes *i ≠ 0*
shows *∃ xs. set xs ⊆ {0..n} ∧ 0 ∉ set xs ∧ i ∉ set xs*
∧ (len M′ 0 0 (i # xs) < 1 ∨ len M′ i i (0 # xs) < 1) ⟨proof⟩

4.8 The Characteristic Property of Canonical DBMs

theorem *fix-index′*:
fixes *M :: (('a :: time) DBMEntry) mat*
assumes *Le r ≤ M i j Le (−r) ≤ M j i cycle-free M n canonical M n i ≤ n j ≤ n i ≠ j*
defines *M′ ≡ λ i′ j′. if (i′ = i ∧ j′ = j) then Le r*
else if (i′ = j ∧ j′ = i) then Le (−r)
else M i′ j′
shows *(∀ u. DBM-val-bounded v u M′ n ⟶ DBM-val-bounded v u M n) ∧ cycle-free M′ n*
⟨proof⟩

lemma *fix-index*:
fixes *M :: (('a :: time) DBMEntry) mat*
assumes *M 0 i + M i 0 > 1 cycle-free M n canonical M n i ≤ n i ≠ 0*
shows
∃ (M′ :: ('a DBMEntry) mat). ((∃ u. DBM-val-bounded v u M′ n) ⟶ (∃ u. DBM-val-bounded v u M n))
∧ M′ 0 i + M′ i 0 = 1 ∧ cycle-free M′ n
∧ (∀ j. i ≠ j ∧ M 0 j + M j 0 = 1 ⟶ M′ 0 j + M′ j 0 = 1)
∧ (∀ j. i ≠ j ∧ M 0 j + M j 0 > 1 ⟶ M′ 0 j + M′ j 0 > 1)
⟨proof⟩

Putting it together **lemma** *FW-not-empty*:
DBM-val-bounded v u (FW M′ n) n ⟹ DBM-val-bounded v u M′ n
⟨proof⟩

lemma *fix-indices*:
fixes $M :: (('a :: time)\ DBMEntry)\ mat$
assumes $set\ xs \subseteq \{0..n\}\ distinct\ xs$
assumes $cyc\text{-}free\ M\ n\ canonical\ M\ n$
shows
$\exists\ (M' :: ('a\ DBMEntry)\ mat).\ ((\exists\ u.\ DBM\text{-}val\text{-}bounded\ v\ u\ M'\ n) \longrightarrow (\exists\ u.\ DBM\text{-}val\text{-}bounded\ v\ u\ M\ n))$
$\wedge\ (\forall\ i \in set\ xs.\ i \neq 0 \longrightarrow M'\ 0\ i + M'\ i\ 0 = \mathbf{1}) \wedge cyc\text{-}free\ M'\ n$
$\wedge\ (\forall\ i{\leq}n.\ i \notin set\ xs \wedge M\ 0\ i + M\ i\ 0 = \mathbf{1} \longrightarrow M'\ 0\ i + M'\ i\ 0 = \mathbf{1})$
⟨*proof*⟩

lemma *cyc-free-obtains-valuation*:
$cyc\text{-}free\ M\ n \Longrightarrow \forall\ c.\ v\ c \leq n \longrightarrow v\ c > 0 \Longrightarrow \exists\ u.\ DBM\text{-}val\text{-}bounded\ v\ u\ M\ n$
⟨*proof*⟩

4.8.1 Floyd-Warshall and Empty DBMs

theorem *FW-detects-empty-zone*:
$\forall\, k{\leq}n.\ 0 < k \longrightarrow (\exists\, c.\ v\ c = k) \Longrightarrow \forall\ c.\ v\ c \leq n \longrightarrow v\ c > 0$
$\Longrightarrow [FW\ M\ n]_{v,n} = \{\} \longleftrightarrow (\exists\ i{\leq}n.\ (FW\ M\ n)\ i\ i < Le\ 0)$
⟨*proof*⟩

hide-const *D*

4.8.2 Mixed Corollaries

lemma *cyc-free-not-empty*:
assumes $cyc\text{-}free\ M\ n\ \forall\, c.\ v\ c \leq n \longrightarrow 0 < v\ c$
shows $[(M :: ('a :: time)\ DBM)]_{v,n} \neq \{\}$
⟨*proof*⟩

lemma *empty-not-cyc-free*:
assumes $\forall\, c.\ v\ c \leq n \longrightarrow 0 < v\ c\ [(M :: ('a :: time)\ DBM)]_{v,n} = \{\}$
shows $\neg\ cyc\text{-}free\ M\ n$
⟨*proof*⟩

lemma *not-empty-cyc-free*:
assumes $\forall\, k{\leq}n.\ 0 < k \longrightarrow (\exists\ c.\ v\ c = k)\ [(M :: ('a :: time)\ DBM)]_{v,n} \neq \{\}$
shows $cyc\text{-}free\ M\ n$ ⟨*proof*⟩

lemma *neg-cycle-empty*:
assumes $\forall\, k{\leq}n.\ 0 < k \longrightarrow (\exists\ c.\ v\ c = k)$ *set xs* $\subseteq \{0..n\}$ $i \leq n$ *len M i i xs* $< \mathbf{1}$
shows $[(M :: ('a :: time)\ DBM)]_{v,n} = \{\}$ ⟨*proof*⟩

abbreviation *clock-numbering'* :: $('c \Rightarrow nat) \Rightarrow nat \Rightarrow bool$
where
clock-numbering' v n $\equiv \forall\ c.\ v\ c > 0 \wedge (\forall\, x.\ \forall\, y.\ v\ x \leq n \wedge v\ y \leq n \wedge v\ x = v\ y \longrightarrow x = y)$

lemma *non-empty-dbm-diag-set*:
clock-numbering' v n $\Longrightarrow [M]_{v,n} \neq \{\} \Longrightarrow [M]_{v,n} = [(\lambda\ i\ j.$ *if* $i = j$ *then* $\mathbf{1}$ *else M i j*$)]_{v,n}$
⟨*proof*⟩

lemma *non-empty-cycle-free*:
assumes $[M]_{v,n} \neq \{\}$
and $\forall\, k{\leq}n.\ 0 < k \longrightarrow (\exists\, c.\ v\ c = k)$
shows *cycle-free M n*
⟨*proof*⟩

lemma *neg-diag-empty*:
assumes $\forall\, k{\leq}n.\ 0 < k \longrightarrow (\exists\, c.\ v\ c = k)$ $i \leq n$ *M i i* $< \mathbf{1}$
shows $[M]_{v,n} = \{\}$
⟨*proof*⟩

lemma *canonical-empty-zone*:
assumes $\forall\, k{\leq}n.\ 0 < k \longrightarrow (\exists\, c.\ v\ c = k)\ \forall\, c.\ v\ c \leq n \longrightarrow 0 < v\ c$
and *canonical M n*
shows $[M]_{v,n} = \{\} \longleftrightarrow (\exists\, i{\leq}n.\ M\ i\ i < \mathbf{1})$
⟨*proof*⟩

end

5 Forward Analysis on DBMs

theory *DBM-Operations*
imports *DBM-Basics*
begin

5.1 Auxiliary

lemma *gt-swap*:
fixes *a b c* :: $'t :: time$

assumes $c < a + b$
shows $c < b + a$
⟨*proof*⟩

lemma *le-swap*:
fixes $a\ b\ c :: 't :: time$
assumes $c \leq a + b$
shows $c \leq b + a$
⟨*proof*⟩

abbreviation *clock-numbering* :: $('c \Rightarrow nat) \Rightarrow bool$
where
clock-numbering $v \equiv \forall\ c.\ v\ c > 0$

5.2 Time Lapse

definition *up* :: $('t::time)\ DBM \Rightarrow ('t::time)\ DBM$
where
up $M \equiv$
$\lambda\ i\ j.$ *if* $i > 0$ *then if* $j = 0$ *then* ∞ *else min* (*dbm-add* $(M\ i\ 0)$ $(M\ 0\ j)$) $(M\ i\ j)$ *else* $M\ i\ j$

lemma *dbm-entry-dbm-lt*:
assumes *dbm-entry-val* u (*Some c1*) (*Some c2*) a $a \prec b$
shows *dbm-entry-val* u (*Some c1*) (*Some c2*) b
⟨*proof*⟩

lemma *dbm-entry-dbm-min2*:
assumes *dbm-entry-val* u *None* (*Some c*) (*min a b*)
shows *dbm-entry-val* u *None* (*Some c*) b
⟨*proof*⟩

lemma *dbm-entry-dbm-min3*:
assumes *dbm-entry-val* u (*Some c*) *None* (*min a b*)
shows *dbm-entry-val* u (*Some c*) *None* b
⟨*proof*⟩

lemma *dbm-entry-dbm-min*:
assumes *dbm-entry-val* u (*Some c1*) (*Some c2*) (*min a b*)
shows *dbm-entry-val* u (*Some c1*) (*Some c2*) b
⟨*proof*⟩

lemma *dbm-entry-dbm-min3'*:
assumes *dbm-entry-val* u (*Some c*) *None* (*min a b*)

shows *dbm-entry-val u* (*Some c*) *None a*
⟨*proof*⟩

lemma *dbm-entry-dbm-min2'*:
assumes *dbm-entry-val u None* (*Some c*) (*min a b*)
shows *dbm-entry-val u None* (*Some c*) *a*
⟨*proof*⟩

lemma *dbm-entry-dbm-min'*:
assumes *dbm-entry-val u* (*Some c1*) (*Some c2*) (*min a b*)
shows *dbm-entry-val u* (*Some c1*) (*Some c2*) *a*
⟨*proof*⟩

lemma *DBM-up-complete'*: *clock-numbering v* $\Longrightarrow$ *u* $\in$ $([M]_{v,n})^{\uparrow}$ $\Longrightarrow$ *u* $\in$ $[up\ M]_{v,n}$
⟨*proof*⟩

fun *theLe* :: ('*t*::*time*) *DBMEntry* $\Rightarrow$ '*t* **where**
theLe (*Le d*) = *d* |
theLe (*Lt d*) = *d* |
theLe ∞ = *0*

lemma *DBM-up-sound'*:
assumes *clock-numbering' v n u* $\in$ $[up\ M]_{v,n}$
shows *u* $\in$ $([M]_{v,n})^{\uparrow}$
⟨*proof*⟩

5.3 From Clock Constraints to DBMs

fun *And* :: ('*t* :: *time*) *DBM* $\Rightarrow$ '*t DBM* $\Rightarrow$ '*t DBM* **where**
And M1 M2 = (λ *i j*. *min* (*M1 i j*) (*M2 i j*))

fun *abstr* :: ('*c*, '*t*::*time*) *cconstraint* $\Rightarrow$ '*t DBM* $\Rightarrow$ ('*c* $\Rightarrow$ *nat*) $\Rightarrow$ '*t DBM* **where**
abstr (*AND cc1 cc2*) *M v* = *And* (*abstr cc1 M v*) (*abstr cc2 M v*) |
abstr (*EQ c d*) *M v* =
(λ *i j* . *if i* = *0* $\wedge$ *j* = *v c then Le* (−*d*) *else if i* = *v c* $\wedge$ *j* = *0 then Le d else M i j*) |
abstr (*LT c d*) *M v* =
(λ *i j* . *if i* = *0* $\wedge$ *j* = *v c then* ∞ *else if i* = *v c* $\wedge$ *j* = *0 then Lt d else M i j*) |
abstr (*LE c d*) *M v* =
(λ *i j* . *if i* = *0* $\wedge$ *j* = *v c then* ∞ *else if i* = *v c* $\wedge$ *j* = *0 then Le d else M i j*) |

abstr (*GT c d*) *M v* =
(λ *i j*. *if i* = *0* ∧ *j* = *v c then Lt* (− *d*) *else if i* = *v c* ∧ *j* = *0 then* ∞ *else M i j*) |
abstr (*GE c d*) *M v* =
(λ *i j*. *if i* = *0* ∧ *j* = *v c then Le* (− *d*) *else if i* = *v c* ∧ *j* = *0 then* ∞ *else M i j*)

lemma *abstr-id1*:
c ∉ *collect-clks cc* ⟹ *clock-numbering′ v n* ⟹ ∀ *c* ∈ *collect-clks cc*. *v c* ≤ *n*
⟹ *abstr cc M v 0* (*v c*) = *M 0* (*v c*)
⟨*proof*⟩

lemma *abstr-id2*:
c ∉ *collect-clks cc* ⟹ *clock-numbering′ v n* ⟹ ∀ *c* ∈ *collect-clks cc*. *v c* ≤ *n*
⟹ *abstr cc M v* (*v c*) *0* = *M* (*v c*) *0*
⟨*proof*⟩

This lemma is trivial because we constrained our theory to difference constraints.

lemma *abstr-id3*:
clock-numbering v ⟹ *abstr cc M v* (*v c1*) (*v c2*) = *M* (*v c1*) (*v c2*)
⟨*proof*⟩

lemma *dbm-abstr-soundness* :
⟦*u* ⊢ *cc*; *clock-numbering′ v n*; ∀ *c* ∈ *collect-clks cc*. *v c* ≤ *n*⟧
⟹ *DBM-val-bounded v u* (*abstr cc* (λ *i j*. ∞) *v*) *n*
⟨*proof*⟩

lemma *dbm-abstr-completeness*:
⟦*DBM-val-bounded v u* (*abstr cc* (λ *i j*. ∞) *v*) *n*; ∀ *c*. *v c* > *0*; ∀ *c* ∈ *collect-clks cc*. *v c* ≤ *n*⟧
⟹ *u* ⊢ *cc*
⟨*proof*⟩

lemma *dbm-abstr-zone-eq*:
assumes *clock-numbering′ v n* ∀ *c*∈*collect-clks cc*. *v c* ≤ *n*
shows [*abstr cc* (λ*i j*. ∞) *v*]$_{v,n}$ = {*u*. *u* ⊢ *cc*}
⟨*proof*⟩

5.4 Zone Intersection

lemma *DBM-and-complete*:

assumes *DBM-val-bounded v u M1 n DBM-val-bounded v u M2 n*
shows *DBM-val-bounded v u (And M1 M2) n*
⟨*proof*⟩

lemma *DBM-and-sound1*:
assumes *DBM-val-bounded v u (And M1 M2) n*
shows *DBM-val-bounded v u M1 n*
⟨*proof*⟩

lemma *DBM-and-sound2*:
assumes *DBM-val-bounded v u (And M1 M2) n*
shows *DBM-val-bounded v u M2 n*
⟨*proof*⟩

5.5 Clock Reset

definition
DBM-reset :: ('*t* :: *time*) *DBM* ⇒ *nat* ⇒ *nat* ⇒ '*t* ⇒ '*t DBM* ⇒ *bool*
where
DBM-reset M n k d M' ≡
(∀ *j* ≤ *n*. 0 < *j* ∧ *k* ≠ *j* ⟶ *M' k j* = ∞ ∧ *M' j k* = ∞) ∧ *M' k* 0 = *Le d* ∧ *M'* 0 *k* = *Le* (− *d*)
∧ *M' k k* = *M k k*
∧ (∀ *i* ≤ *n*. ∀ *j* ≤ *n*.
i ≠ *k* ∧ *j* ≠ *k* ⟶ *M' i j* = *min* (*dbm-add* (*M i k*) (*M k j*)) (*M i j*))

lemma *DBM-reset-mono*:
assumes *DBM-reset M n k d M' i* ≤ *n j* ≤ *n i* ≠ *k j* ≠ *k*
shows *M' i j* ≤ *M i j*
⟨*proof*⟩

lemma *DBM-reset-len-mono*:
assumes *DBM-reset M n k d M' k* ∉ *set xs i* ≠ *k j* ≠ *k set* (*i* # *j* # *xs*) ⊆ {0..*n*}
shows *len M' i j xs* ≤ *len M i j xs*
⟨*proof*⟩

lemma *DBM-reset-neg-cycle-preservation*:
assumes *DBM-reset M n k d M' len M i i xs* < *Le* 0 *set* (*k* # *i* # *xs*) ⊆ {0..*n*}
shows ∃ *j*. ∃ *ys*. *set* (*j* # *ys*) ⊆ {0..*n*} ∧ *len M' j j ys* < *Le* 0
⟨*proof*⟩

Implementation of DBM reset

definition *reset* :: ('t::*time*) *DBM* ⇒ *nat* ⇒ *nat* ⇒ 't ⇒ 't *DBM*
where
reset M n k d =
(λ *i j*.
if i = *k* ∧ *j* = *0 then Le d*
else if i = *0* ∧ *j* = *k then Le* (−*d*)
else if i = *k* ∧ *j* ≠ *k then* ∞
else if i ≠ *k* ∧ *j* = *k then* ∞
else if i = *k* ∧ *j* = *k then M k k*
else min (*dbm-add* (*M i k*) (*M k j*)) (*M i j*)
)

fun *reset'* :: ('t::*time*) *DBM* ⇒ *nat* ⇒ 'c *list* ⇒ ('c ⇒ *nat*) ⇒ 't ⇒ 't *DBM*
where
reset' M n [] *v d* = *M* |
reset' M n (*c* # *cs*) *v d* = *reset* (*reset' M n cs v d*) *n* (*v c*) *d*

lemma *DBM-reset-reset*:
0 < *k* ⟹ *k* ≤ *n* ⟹ *DBM-reset M n k d* (*reset M n k d*)
⟨*proof*⟩

lemma *DBM-reset-complete*:
assumes *clock-numbering' v n v c* ≤ *n DBM-reset M n* (*v c*) *d M'*
DBM-val-bounded v u M n
shows *DBM-val-bounded v* (*u*(*c* := *d*)) *M' n*
⟨*proof*⟩

lemma *DBM-reset-sound-empty*:
assumes *clock-numbering' v n v c* ≤ *n DBM-reset M n* (*v c*) *d M'*
∀ *u* . ¬ *DBM-val-bounded v u M' n*
shows ¬ *DBM-val-bounded v u M n*
⟨*proof*⟩

lemma *DBM-reset-diag-preservation*:
∀ *k*≤*n*. *M k k* ≤ **1** ⟹ *DBM-reset M n i d M'* ⟹ ∀ *k*≤*n*. *M' k k* ≤ **1**
⟨*proof*⟩

lemma *FW-diag-preservation*:
∀ *k*≤*n*. *M k k* ≤ **1** ⟹ ∀ *k*≤*n*. (*FW M n*) *k k* ≤ **1**
⟨*proof*⟩

lemma *DBM-reset-not-cyc-free-preservation*:

assumes ¬ *cyc-free M n DBM-reset M n k d M′ k* ≤ *n*
shows ¬ *cyc-free M′ n*
⟨*proof*⟩

lemma *DBM-reset-complete-empty′*:
assumes ∀ *k*≤*n*. *k* > *0* ⟶ (∃ *c*. *v c* = *k*) *clock-numbering v k* ≤ *n*
DBM-reset M n k d M′ ∀ *u* . ¬ *DBM-val-bounded v u M n*
shows ¬ *DBM-val-bounded v u M′ n*
⟨*proof*⟩

lemma *DBM-reset-complete-empty*:
assumes ∀ *k*≤*n*. *k* > *0* ⟶ (∃ *c*. *v c* = *k*) *clock-numbering v*
DBM-reset (*FW M n*) *n* (*v c*) *d M′* ∀ *u* . ¬ *DBM-val-bounded v u* (*FW M n*) *n*
shows ¬ *DBM-val-bounded v u M′ n*
⟨*proof*⟩

lemma *DBM-reset-complete-empty1*:
assumes ∀ *k*≤*n*. *k* > *0* ⟶ (∃ *c*. *v c* = *k*) *clock-numbering v*
DBM-reset (*FW M n*) *n* (*v c*) *d M′* ∀ *u* . ¬ *DBM-val-bounded v u M n*
shows ¬ *DBM-val-bounded v u M′ n*
⟨*proof*⟩

Lemma *FW-canonical-id* allows us to prove correspondences between reset and canonical, like for the two below. Can be left out for the rest because of the triviality of the correspondence.

lemma *DBM-reset-empty″*:
assumes ∀ *k*≤*n*. *k* > *0* ⟶ (∃ *c*. *v c* = *k*) *clock-numbering′ v n v c* ≤ *n*
DBM-reset M n (*v c*) *d M′*
shows $[M]_{v,n} = \{\} \longleftrightarrow [M']_{v,n} = \{\}$
⟨*proof*⟩

lemma *DBM-reset-empty*:
assumes ∀ *k*≤*n*. *k* > *0* ⟶ (∃ *c*. *v c* = *k*) *clock-numbering′ v n v c* ≤ *n*
DBM-reset (*FW M n*) *n* (*v c*) *d M′*
shows $[FW\ M\ n]_{v,n} = \{\} \longleftrightarrow [M']_{v,n} = \{\}$
⟨*proof*⟩

lemma *DBM-reset-empty′*:
assumes *canonical M n* ∀ *k*≤*n*. *k* > *0* ⟶ (∃ *c*. *v c* = *k*) *clock-numbering′ v n v c* ≤ *n*
DBM-reset (*FW M n*) *n* (*v c*) *d M′*
shows $[M]_{v,n} = \{\} \longleftrightarrow [M']_{v,n} = \{\}$

⟨*proof*⟩

lemma *DBM-reset-sound′*:
assumes *clock-numbering′ v n v c* ≤ *n DBM-reset M n* (*v c*) *d M′ DBM-val-bounded v u M′ n*
DBM-val-bounded v u″ M n
obtains *d′* **where** *DBM-val-bounded v* (*u*(*c* := *d′*)) *M n*
⟨*proof*⟩

lemma *DBM-reset-sound2*:
assumes *v c* ≤ *n DBM-reset M n* (*v c*) *d M′ DBM-val-bounded v u M′ n*
shows *u c* = *d*
⟨*proof*⟩

lemma *DBM-reset-sound″*:
fixes *M v c n d*
defines *M′* ≡ *reset M n* (*v c*) *d*
assumes *clock-numbering′ v n v c* ≤ *n DBM-val-bounded v u M′ n*
DBM-val-bounded v u″ M n
obtains *d′* **where** *DBM-val-bounded v* (*u*(*c* := *d′*)) *M n*
⟨*proof*⟩

lemma *DBM-reset-sound*:
fixes *M v c n d*
defines *M′* ≡ *reset M n* (*v c*) *d*
assumes ∀ *k*≤*n*. *k* > *0* ⟶ (∃ *c*. *v c* = *k*) *clock-numbering′ v n v c* ≤ *n*
u ∈ ⌈*M′*⌉$_{v,n}$
obtains *d′* **where** *u*(*c* := *d′*) ∈[*M*]$_{v,n}$
⟨*proof*⟩

lemma *DBM-reset′-complete′*:
assumes *DBM-val-bounded v u M n clock-numbering′ v n* ∀ *c* ∈ *set cs. v c* ≤ *n*
shows ∃ *u′*. *DBM-val-bounded v u′* (*reset′ M n cs v d*) *n*
⟨*proof*⟩

lemma *DBM-reset′-complete*:
assumes *DBM-val-bounded v u M n clock-numbering′ v n* ∀ *c* ∈ *set cs. v c* ≤ *n*
shows *DBM-val-bounded v* ([*cs* → *d*]*u*) (*reset′ M n cs v d*) *n*
⟨*proof*⟩

lemma *DBM-reset′-sound-empty*:
assumes *clock-numbering′ v n* ∀ *c* ∈ *set cs. v c* ≤ *n*

$\forall\ u\ .\ \neg$ *DBM-val-bounded* $v\ u$ (*reset'* $M\ n\ cs\ v\ d$) n
shows $\neg$ *DBM-val-bounded* $v\ u\ M\ n$
⟨*proof*⟩

fun *set-clocks* :: $'c$ *list* $\Rightarrow$ $'t$::*time list*$\Rightarrow$ ($'c$,$'t$) *cval* $\Rightarrow$ ($'c$,$'t$) *cval*
where
set-clocks [] - $u = u$ |
set-clocks - [] $u = u$ |
set-clocks (c#cs) (t#ts) u = (*set-clocks* $cs\ ts$ (u(c:=t)))

lemma *DBM-reset'-sound'*:
fixes $M\ v\ c\ n\ d\ cs$
assumes *clock-numbering'* $v\ n\ \forall\ c \in$ *set* $cs.\ v\ c \leq n$
DBM-val-bounded $v\ u$ (*reset'* $M\ n\ cs\ v\ d$) n *DBM-val-bounded* $v\ u''$ $M\ n$
shows $\exists\, ts.$ *DBM-val-bounded* v (*set-clocks* $cs\ ts\ u$) $M\ n$
⟨*proof*⟩

lemma *DBM-reset'-resets*:
fixes $M\ v\ c\ n\ d\ cs$
assumes $\forall\, k{\leq}n.\ k > 0 \longrightarrow (\exists\, c.\ v\ c = k)$ *clock-numbering'* $v\ n\ \forall\ c \in$ *set* $cs.\ v\ c \leq n$
DBM-val-bounded $v\ u$ (*reset'* $M\ n\ cs\ v\ d$) n
shows $\forall\, c \in$ *set* $cs.\ u\ c = d$
⟨*proof*⟩

lemma *DBM-reset'-resets'*:
fixes $M\ v\ c\ n\ d\ cs$
assumes *clock-numbering'* $v\ n\ \forall\ c \in$ *set* $cs.\ v\ c \leq n$ *DBM-val-bounded* $v\ u$ (*reset'* $M\ n\ cs\ v\ d$) n
DBM-val-bounded $v\ u''\ M\ n$
shows $\forall\, c \in$ *set* $cs.\ u\ c = d$
⟨*proof*⟩

lemma *DBM-reset'-neg-diag-preservation'*:
assumes $k{\leq}n\ M\ k\ k < \mathbf{1}$ *clock-numbering* $v\ \forall\ c \in$ *set* $cs.\ v\ c \leq n$
shows *reset'* $M\ n\ cs\ v\ d\ k\ k < \mathbf{1}$ ⟨*proof*⟩

lemma *DBM-reset'-complete-empty'*:
assumes $\forall\, k{\leq}n.\ k > 0 \longrightarrow (\exists\, c.\ v\ c = k)$ *clock-numbering'* $v\ n$
$\forall\ c \in$ *set* $cs.\ v\ c \leq n\ \forall\ u\ .\ \neg$ *DBM-val-bounded* $v\ u\ M\ n$
shows $\forall\ u\ .\ \neg$ *DBM-val-bounded* $v\ u$ (*reset'* $M\ n\ cs\ v\ d$) n ⟨*proof*⟩

lemma *DBM-reset'-complete-empty*:

assumes $\forall\, k{\leq}n.\ k > 0 \longrightarrow (\exists\, c.\ v\ c = k)$ *clock-numbering'* $v\ n$
$\forall\ c \in$ *set cs.* $v\ c \leq n \forall\ u\ .\ \neg$ *DBM-val-bounded* $v\ u\ M\ n$
shows $\forall\ u\ .\ \neg$ *DBM-val-bounded* $v\ u$ (*reset'* (*FW* $M\ n$) n *cs* $v\ d$) n ⟨*proof*⟩

lemma *DBM-reset'-empty'*:
assumes $\forall\, k{\leq}n.\ k > 0 \longrightarrow (\exists\, c.\ v\ c = k)$ *clock-numbering'* $v\ n \forall\ c \in$ *set cs.* $v\ c \leq n$
shows $[M]_{v,n} = \{\} \longleftrightarrow$ [*reset'* (*FW* $M\ n$) n *cs* $v\ d]_{v,n} = \{\}$
⟨*proof*⟩

lemma *DBM-reset'-empty*:
assumes $\forall\, k{\leq}n.\ k > 0 \longrightarrow (\exists\, c.\ v\ c = k)$ *clock-numbering'* $v\ n \forall\ c \in$ *set cs.* $v\ c \leq n$
shows $[M]_{v,n} = \{\} \longleftrightarrow$ [*reset'* $M\ n$ *cs* $v\ d]_{v,n} = \{\}$
⟨*proof*⟩

lemma *DBM-reset'-sound*:
assumes $\forall\, k{\leq}n.\ k > 0 \longrightarrow (\exists\, c.\ v\ c = k)$ *clock-numbering'* $v\ n$
and $\forall\, c{\in}$*set cs.* $v\ c \leq n$
and $u \in$ [*reset'* $M\ n$ *cs* $v\ d]_{v,n}$
shows $\exists$ *ts. set-clocks cs ts* $u \in [M]_{v,n}$
⟨*proof*⟩

5.6 Misc Preservation Lemmas

lemma *get-const-sum*[*simp*]:
$a \neq \infty \Longrightarrow b \neq \infty \Longrightarrow$ *get-const* $a \in \mathbb{Z} \Longrightarrow$ *get-const* $b \in \mathbb{Z} \Longrightarrow$ *get-const* $(a + b) \in \mathbb{Z}$
⟨*proof*⟩

lemma *sum-not-inf-dest*:
assumes $a + b \neq \infty$
shows $a \neq \infty \wedge b \neq \infty$
⟨*proof*⟩

lemma *sum-not-inf-int*:
assumes $a + b \neq \infty$ *get-const* $a \in \mathbb{Z}$ *get-const* $b \in \mathbb{Z}$
shows *get-const* $(a + b) \in \mathbb{Z}$
⟨*proof*⟩

lemma *int-fw-upd*:
$\forall\ i \leq n. \forall\ j \leq n.\ m\ i\ j \neq \infty \longrightarrow$ *get-const* $(m\ i\ j) \in \mathbb{Z} \Longrightarrow k \leq n \Longrightarrow i \leq n \Longrightarrow j \leq n$
$\Longrightarrow i' \leq n \Longrightarrow j' \leq n \Longrightarrow$ (*fw-upd* $m\ k\ i\ j\ i'\ j'$) $\neq \infty$

$\Longrightarrow$ *get-const* (*fw-upd m k i j i' j'*) $\in \mathbb{Z}$
⟨*proof*⟩

lemma *fw-int-aux-c*:
assumes $\forall\ i \leq n.\ \forall\ j \leq n.\ M\ i\ j \neq \infty \longrightarrow$ *get-const* $(M\ i\ j) \in \mathbb{Z}$ $a \leq n$ $b \leq n$ $c \leq n$
$i \leq n$ $j \leq n$ $((fw\ M\ n)\ 0\ 0\ c)\ i\ j \neq \infty$
shows *get-const* $(((fw\ M\ n)\ 0\ 0\ c)\ i\ j) \in \mathbb{Z}$
⟨*proof*⟩

lemma *fw-int-aux-Suc-b*:
assumes $\forall\ i \leq n.\ \forall\ j \leq n.\ (fw\ M\ n)\ a\ b\ n\ i\ j \neq \infty \longrightarrow$ *get-const* $((fw\ M\ n)\ a\ b\ n\ i\ j) \in \mathbb{Z}$
$a \leq n$ $Suc\ b \leq n$ $c \leq n$ $i \leq n$ $j \leq n$ $((fw\ M\ n)\ a\ (Suc\ b)\ c)\ i\ j \neq \infty$
shows *get-const* $(((fw\ M\ n)\ a\ (Suc\ b)\ c)\ i\ j) \in \mathbb{Z}$
⟨*proof*⟩

lemma *fw-int-aux-b*:
assumes $\forall\ i \leq n.\ \forall\ j \leq n.\ M\ i\ j \neq \infty \longrightarrow$ *get-const* $(M\ i\ j) \in \mathbb{Z}$ $a \leq n$ $b \leq n$ $c \leq n$
$i \leq n$ $j \leq n$ $((fw\ M\ n)\ 0\ b\ c)\ i\ j \neq \infty$
shows *get-const* $(((fw\ M\ n)\ 0\ b\ c)\ i\ j) \in \mathbb{Z}$ ⟨*proof*⟩

lemma *fw-int-aux-Suc-a*:
assumes $\forall\ i \leq n.\ \forall\ j \leq n.\ (fw\ M\ n)\ a\ n\ n\ i\ j \neq \infty \longrightarrow$ *get-const* $((fw\ M\ n)\ a\ n\ n\ i\ j) \in \mathbb{Z}$
$Suc\ a \leq n$ $b \leq n$ $c \leq n$ $i \leq n$ $j \leq n$ $((fw\ M\ n)\ (Suc\ a)\ b\ c)\ i\ j \neq \infty$
shows *get-const* $(((fw\ M\ n)\ (Suc\ a)\ b\ c)\ i\ j) \in \mathbb{Z}$
⟨*proof*⟩

lemma *fw-int-preservation*:
assumes $\forall\ i \leq n.\ \forall\ j \leq n.\ M\ i\ j \neq \infty \longrightarrow$ *get-const* $(M\ i\ j) \in \mathbb{Z}$ $a \leq n$ $b \leq n$ $c \leq n$
$i \leq n$ $j \leq n$ $((fw\ M\ n)\ a\ b\ c)\ i\ j \neq \infty$
shows *get-const* $(((fw\ M\ n)\ a\ b\ c)\ i\ j) \in \mathbb{Z}$
⟨*proof*⟩

lemma *FW-int-preservation*:
assumes $\forall\ i \leq n.\ \forall\ j \leq n.\ M\ i\ j \neq \infty \longrightarrow$ *get-const* $(M\ i\ j) \in \mathbb{Z}$
shows $\forall\ i \leq n.\ \forall\ j \leq n.\ FW\ M\ n\ i\ j \neq \infty \longrightarrow$ *get-const* $(FW\ M\ n\ i\ j) \in \mathbb{Z}$
⟨*proof*⟩

abbreviation *dbm-int* $M\ n \equiv \forall\ i{\leq}n.\ \forall\ j{\leq}n.\ M\ i\ j \neq \infty \longrightarrow$ *get-const* $(M$

$i\ j) \in \mathbb{Z}$

lemma *And-int-preservation*:
assumes *dbm-int M1 n dbm-int M2 n*
shows *dbm-int* (*And M1 M2*) *n*
⟨*proof*⟩

lemma *up-int-preservation*:
dbm-int M n $\Longrightarrow$ *dbm-int* (*up M*) *n*
⟨*proof*⟩

lemma *DBM-reset-int-preservation′*:
assumes *dbm-int M n DBM-reset M n k d M′* $d \in \mathbb{Z}$ $k \leq n$
shows *dbm-int M′ n*
⟨*proof*⟩

lemma *DBM-reset-int-preservation*:
assumes *dbm-int M n* $d \in \mathbb{Z}$ $0 < k$ $k \leq n$
shows *dbm-int* (*reset M n k d*) *n*
⟨*proof*⟩

lemma *DBM-reset′-int-preservation*:
assumes *dbm-int M n* $d \in \mathbb{Z}$ $\forall\, c.\ v\ c > 0\ \forall\ c \in$ *set cs. v c* $\leq n$
shows *dbm-int* (*reset′ M n cs v d*) *n* ⟨*proof*⟩

lemma *int-zone-dbm*:
assumes *clock-numbering′ v n*
$\forall$ (-,*d*) $\in$ *collect-clock-pairs cc. d* $\in \mathbb{Z}$ $\forall$ *c* $\in$ *collect-clks cc. v c* $\leq n$
obtains *M* **where** $\{u.\ u \vdash cc\} = [M]_{v,n}$
and $\forall\ i \leq n.\ \forall\ j \leq n.\ M\ i\ j \neq \infty \longrightarrow$ *get-const* (*M i j*) $\in \mathbb{Z}$
⟨*proof*⟩

lemma *reset-set1*:
$\forall\, c \in$ *set cs.* ([*cs*→*d*]*u*) *c* = *d*
⟨*proof*⟩

lemma *reset-set11*:
$\forall\, c.\ c \notin$ *set cs* $\longrightarrow$ ([*cs*→*d*]*u*) *c* = *u c*
⟨*proof*⟩

lemma *reset-set2*:
$\forall\, c.\ c \notin$ *set cs* $\longrightarrow$ (*set-clocks cs ts u*)*c* = *u c*
⟨*proof*⟩

lemma *reset-set*:
assumes $\forall\ c \in set\ cs.\ u\ c = d$
shows $[cs \rightarrow d](set\text{-}clocks\ cs\ ts\ u) = u$
⟨*proof*⟩

abbreviation *global-clock-numbering* ::
$('a, 'c, 't :: time, 's)\ ta \Rightarrow ('c \Rightarrow nat) \Rightarrow nat \Rightarrow bool$
where
global-clock-numbering A v n ≡
clock-numbering′ v n ∧ (∀ $c \in clk\text{-}set\ A.\ v\ c \leq n$) ∧ (∀ $k \leq n.\ k > 0 \longrightarrow$
(∃ $c.\ v\ c = k$))

lemma *dbm-int-abstr*:
assumes ∀ $(x, m) \in collect\text{-}clock\text{-}pairs\ g.\ m \in \mathbb{Z}$
shows *dbm-int* $(abstr\ g\ (\lambda i\ j.\ \infty)\ v)\ n$
⟨*proof*⟩

lemma *dbm-int-inv-abstr*:
assumes ∀ $(x,m) \in clkp\text{-}set\ A.\ m \in \mathbb{N}$
shows *dbm-int* $(abstr\ (inv\text{-}of\ A\ l)\ (\lambda i\ j.\ \infty)\ v)\ n$
⟨*proof*⟩

lemma *dbm-int-guard-abstr*:
assumes *valid-abstraction A X k A* ⊢ $l \longrightarrow^{g,a,r} l'$
shows *dbm-int* $(abstr\ g\ (\lambda i\ j.\ \infty)\ v)\ n$
⟨*proof*⟩

lemma *collect-clks-id*: *collect-clks cc* = *fst* ‘ *collect-clock-pairs cc* ⟨*proof*⟩

5.6.1 Unused theorems

lemma *canonical-cyc-free*:
canonical M n ⟹ ∀ $i \leq n.\ M\ i\ i \geq \mathbf{1}$ ⟹ *cyc-free M n*
⟨*proof*⟩

lemma *canonical-cyc-free2*:
canonical M n ⟹ *cyc-free M n* ⟷ (∀ $i \leq n.\ M\ i\ i \geq \mathbf{1}$)
⟨*proof*⟩

lemma *DBM-reset′-diag-preservation*:
assumes ∀ $k \leq n.\ M\ k\ k \leq \mathbf{1}$ *clock-numbering v* ∀ $c \in set\ cs.\ v\ c \leq n$
shows ∀ $k \leq n.\ reset'\ M\ n\ cs\ v\ d\ k\ k \leq \mathbf{1}$ ⟨*proof*⟩

end

5.6.2 Semantics Based on DBMs

theory *DBM-Zone-Semantics*
imports *DBM-Operations*
begin

5.6.3 Single Step

inductive *step-z-dbm* ::
(′*a*, ′*c*, ′*t*, ′*s*) *ta* ⇒ ′*s* ⇒ (′*t*::*time*) *DBM*
⇒ (′*c* ⇒ *nat*) ⇒ *nat* ⇒ ′*s* ⇒ (′*t*::*time*) *DBM* ⇒ *bool*
(- ⊢ ⟨-, -⟩ ⇝-,- ⟨-, -⟩ [*61*,*61*,*61*] *61*)
where
step-t-z-dbm:
D-inv = *abstr* (*inv-of A l*) (λ*i j*. ∞) *v* ⟹ *A* ⊢ ⟨*l*,*D*⟩ ⇝*v*,*n* ⟨*l*,*And* (*up* (*And D D-inv*)) *D-inv*⟩ |
step-a-z-dbm:
A ⊢ *l* ⟶*g*,*a*,*r* *l*′
⟹ *A* ⊢ ⟨*l*,*D*⟩ ⇝*v*,*n* ⟨*l*′,*And* (*reset*′ (*And D* (*abstr g* (λ*i j*. ∞) *v*)) *n r v* *0*)
(*abstr* (*inv-of A l*′) (λ*i j*. ∞) *v*)⟩
inductive-cases *step-z-cases*: *A* ⊢ ⟨*l*, *D*⟩ ⇝*v*,*n* ⟨*l*′, *D*′⟩

declare *step-z-dbm.intros*[*intro*]

lemma *step-z-dbm-preserves-int*:
assumes *A* ⊢ ⟨*l*,*D*⟩ ⇝*v*,*n* ⟨*l*′,*D*′⟩ *global-clock-numbering A v n valid-abstraction A X k*
dbm-int D n
shows *dbm-int D*′ *n*
⟨*proof*⟩

lemma *And-correct*:
shows [*M1*]*v*,*n* ∩ [*M2*]*v*,*n* = [*And M1 M2*]*v*,*n*
⟨*proof*⟩

lemma *up-correct*:
assumes *clock-numbering*′ *v n*
shows [*up M*]*v*,*n* = [*M*]*v*,*n*↑
⟨*proof*⟩

lemma *step-z-dbm-sound*:

assumes $A \vdash \langle l,\ D\rangle \rightsquigarrow_{v,n} \langle l',\ D'\rangle$ *global-clock-numbering* $A\ v\ n$
shows $A \vdash \langle l,\ [D]_{v,n}\rangle \rightsquigarrow \langle l',\ [D']_{v,n}\rangle$
⟨*proof*⟩

lemma *step-z-dbm-DBM*:
assumes $A \vdash \langle l,\ [D]_{v,n}\rangle \rightsquigarrow \langle l',\ Z\rangle$ *global-clock-numbering* $A\ v\ n$
obtains D' **where** $A \vdash \langle l,\ D\rangle \rightsquigarrow_{v,n} \langle l',\ D'\rangle\ Z = [D']_{v,n}$
⟨*proof*⟩

lemma *step-z-computable*:
assumes $A \vdash \langle l,\ [D]_{v,n}\rangle \rightsquigarrow \langle l',Z\rangle$ *global-clock-numbering* $A\ v\ n$
obtains D' **where** $Z = [D']_{v,n}$
⟨*proof*⟩

lemma *step-z-dbm-complete*:
assumes *global-clock-numbering* $A\ v\ n$ $A \vdash \langle l,\ u\rangle \rightarrow \langle l',u'\rangle$
and $u \in [(D\ )]_{v,n}$
shows $\exists\ D'.\ A \vdash \langle l,\ D\rangle \rightsquigarrow_{v,n} \langle l',D'\rangle \wedge u' \in [D']_{v,n}$
⟨*proof*⟩

5.6.4 Multi Step

inductive *steps-z-dbm* ::
$('a,\ 'c,\ 't,\ 's)\ ta \Rightarrow 's \Rightarrow ('t::time)\ DBM$
$\Rightarrow ('c \Rightarrow nat) \Rightarrow nat \Rightarrow 's \Rightarrow ('t::time)\ DBM \Rightarrow bool$
(- ⊢ ⟨-, -⟩ ⇝*-,- ⟨-, -⟩ [61,61,61] 61)
where
refl: $A \vdash \langle l,D\rangle \rightsquigarrow *_{v,n} \langle l,D\rangle$ |
step: $A \vdash \langle l,D\rangle \rightsquigarrow_{v,n} \langle l',D'\rangle \Longrightarrow A \vdash \langle l',D'\rangle \rightsquigarrow *_{v,n} \langle l'',D''\rangle \Longrightarrow$
$A \vdash \langle l,D\rangle \rightsquigarrow *_{v,n} \langle l'',D''\rangle$

declare *steps-z-dbm.intros*[*intro*]

lemma *steps-z-dbm-sound*:
assumes $A \vdash \langle l,D\rangle \rightsquigarrow *_{v,n} \langle l',D'\rangle$
and *global-clock-numbering* $A\ v\ n$
and $u' \in [D']_{v,n}$
shows $\exists\ u \in [D]_{v,n}.\ A \vdash \langle l,\ u\rangle \rightarrow * \langle l',u'\rangle$ ⟨*proof*⟩

lemma *steps-z-dbm-complete*:
assumes $A \vdash \langle l,\ u\rangle \rightarrow * \langle l',u'\rangle$
and *global-clock-numbering* $A\ v\ n$
and $u \in [D]_{v,n}$
shows $\exists\ D'.\ A \vdash \langle l,\ D\rangle \rightsquigarrow *_{v,n} \langle l',\ D'\rangle \wedge u' \in [D']_{v,n}$ ⟨*proof*⟩

end

⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*

5.7 Normalization of DBMs

theory *DBM-Normalization*
imports *DBM-Basics*
begin

This is the implementation of the common approximation operation.

fun *norm-upper* :: ('*t*::*time*) *DBMEntry* ⇒ '*t* ⇒ ('*t*::*time*) *DBMEntry*
where
norm-upper e t = (*if Le t* ≺ *e then* ∞ *else e*)

fun *norm-lower* :: ('*t*::*time*) *DBMEntry* ⇒ '*t* ⇒ ('*t*::*time*) *DBMEntry*
where
norm-lower e t = (*if e* ≺ *Lt t then Lt t else e*)

Note that literature pretends that **0** would have some (presumably infinite bound) in k and thus defines normalization uniformly. The easiest way to get around this seems to explicate this in the definition as below.

definition *norm* :: ('*t*::*time*) *DBM* ⇒ (*nat* ⇒ '*t*) ⇒ *nat* ⇒ '*t DBM*
where
norm M k n ≡ λ *i j*.
let ub = *if i* > *0 then* (*k i*) *else 0 in*
let lb = *if j* > *0 then* (− *k j*) *else 0 in*
if i ≤ *n* ∧ *j* ≤ *n then norm-lower* (*norm-upper* (*M i j*) *ub*) *lb else M i j*

5.8 Normalization is a Widening Operator

lemma *norm-mono*:
assumes ∀ *c*. *v c* > *0 u* ∈ $[M]_{v,n}$
shows *u* ∈ [*norm M k n*]$_{v,n}$ (**is** *u* ∈ [*?M2*]$_{v,n}$)
⟨*proof*⟩

end
theory *Regions-Beta*
imports *Misc DBM-Normalization DBM-Operations*
begin

6 Refinement to β-regions

6.1 Definition

type-synonym *'c ceiling* = (*'c* ⇒ *nat*)

datatype *intv* =
  Const nat |
  Intv nat |
  Greater nat

datatype *intv'* =
  Const' int |
  Intv' int |
  Greater' int |
  Smaller' int

type-synonym *t* = *real*

instantiation *real* :: *time*
begin
  instance ⟨*proof*⟩
end

inductive *valid-intv* :: *nat* ⇒ *intv* ⇒ *bool*
where
  0 ≤ *d* ⟹ *d* ≤ *c* ⟹ *valid-intv c* (*Const d*) |
  0 ≤ *d* ⟹ *d* < *c* ⟹ *valid-intv c* (*Intv d*) |
  valid-intv c (*Greater c*)

inductive *valid-intv'* :: *int* ⇒ *int* ⇒ *intv'* ⇒ *bool*
where
  valid-intv' l - (*Smaller'* (−*l*)) |
  −*l* ≤ *d* ⟹ *d* ≤ *u* ⟹ *valid-intv' l u* (*Const' d*) |
  −*l* ≤ *d* ⟹ *d* < *u* ⟹ *valid-intv' l u* (*Intv' d*) |
  valid-intv' - *u* (*Greater' u*)

inductive *intv-elem* :: *'c* ⇒ (*'c*,*t*) *cval* ⇒ *intv* ⇒ *bool*
where
  u x = *d* ⟹ *intv-elem x u* (*Const d*) |
  d < *u x* ⟹ *u x* < *d* + *1* ⟹ *intv-elem x u* (*Intv d*) |
  c < *u x* ⟹ *intv-elem x u* (*Greater c*)

inductive *intv'-elem* :: *'c* ⇒ *'c* ⇒ (*'c*,*t*) *cval* ⇒ *intv'* ⇒ *bool*

where
$u\ x - u\ y < c \Longrightarrow intv'\text{-}elem\ x\ y\ u\ (Smaller'\ c)$ |
$u\ x - u\ y = d \Longrightarrow intv'\text{-}elem\ x\ y\ u\ (Const'\ d)$ |
$d < u\ x - u\ y \Longrightarrow u\ x - u\ y < d + 1 \Longrightarrow intv'\text{-}elem\ x\ y\ u\ (Intv'\ d)$ |
$c < u\ x - u\ y \Longrightarrow intv'\text{-}elem\ x\ y\ u\ (Greater'\ c)$

abbreviation *total-preorder r* ≡ *refl r* ∧ *trans r*

inductive *isConst* :: *intv* ⇒ *bool*
where
isConst (*Const* -)

inductive *isIntv* :: *intv* ⇒ *bool*
where
isIntv (*Intv* -)

inductive *isGreater* :: *intv* ⇒ *bool*
where
isGreater (*Greater* -)

declare *isIntv.intros*[*intro*!] *isConst.intros*[*intro*!] *isGreater.intros*[*intro*!]

declare *isIntv.cases*[*elim*!] *isConst.cases*[*elim*!] *isGreater.cases*[*elim*!]

inductive *valid-region* :: *'c set* ⇒ (*'c* ⇒ *nat*) ⇒ (*'c* ⇒ *intv*) ⇒ (*'c* ⇒ *'c* ⇒ *intv'*) ⇒ *'c rel* ⇒ *bool*
where
⟦$X_0 = \{x \in X.\ \exists\ d.\ I\ x = Intv\ d\}$; *refl-on* X_0 *r*; *trans r*; *total-on* X_0 *r*; ∀ *x* ∈ *X*. *valid-intv* (*k x*) (*I x*);
∀ *x* ∈ *X*. ∀ *y* ∈ *X*. *isGreater* (*I x*) ∨ *isGreater* (*I y*) ⟶ *valid-intv'* (*k y*) (*k x*) (*J x y*)⟧
⟹ *valid-region X k I J r*

inductive-set *region* **for** *X I J r*
where
∀ *x* ∈ *X*. *u x* ≥ *0* ⟹ ∀ *x* ∈ *X*. *intv-elem x u* (*I x*) ⟹ $X_0 = \{x \in X.\ \exists\ d.\ I\ x = Intv\ d\}$ ⟹
∀ *x* ∈ X_0. ∀ *y* ∈ X_0. (*x*, *y*) ∈ *r* ⟷ *frac* (*u x*) ≤ *frac* (*u y*) ⟹
∀ *x* ∈ *X*. ∀ *y* ∈ *X*. *isGreater* (*I x*) ∨ *isGreater* (*I y*) ⟶ *intv'-elem x y u* (*J x y*)
⟹ *u* ∈ *region X I J r*

Defining the unique element of a partition that contains a valuation

definition *part* ([-]- [*61*,*61*] *61*) **where** *part v* $\mathcal{R}$ ≡ *THE R*. *R* ∈ $\mathcal{R}$ ∧ *v* ∈

R

First we need to show that the set of regions is a partition of the set of all clock assignments. This property is only claimed by P. Bouyer.

inductive-cases[*elim*!]: *intv-elem x u* (*Const d*)
inductive-cases[*elim*!]: *intv-elem x u* (*Intv d*)
inductive-cases[*elim*!]: *intv-elem x u* (*Greater d*)
inductive-cases[*elim*!]: *valid-intv c* (*Greater d*)
inductive-cases[*elim*!]: *valid-intv c* (*Const d*)
inductive-cases[*elim*!]: *valid-intv c* (*Intv d*)
inductive-cases[*elim*!]: *intv′-elem x y u* (*Const′ d*)
inductive-cases[*elim*!]: *intv′-elem x y u* (*Intv′ d*)
inductive-cases[*elim*!]: *intv′-elem x y u* (*Greater′ d*)
inductive-cases[*elim*!]: *intv′-elem x y u* (*Smaller′ d*)
inductive-cases[*elim*!]: *valid-intv′ l u* (*Greater′ d*)
inductive-cases[*elim*!]: *valid-intv′ l u* (*Smaller′ d*)
inductive-cases[*elim*!]: *valid-intv′ l u* (*Const′ d*)
inductive-cases[*elim*!]: *valid-intv′ l u* (*Intv′ d*)

declare *valid-intv.intros*[*intro*]
declare *valid-intv′.intros*[*intro*]
declare *intv-elem.intros*[*intro*]
declare *intv′-elem.intros*[*intro*]

declare *region.cases*[*elim*]
declare *valid-region.cases*[*elim*]

6.2 Basic Properties

First we show that all valid intervals are distinct

lemma *valid-intv-distinct*:
valid-intv c I $\Longrightarrow$ *valid-intv c I′* $\Longrightarrow$ *intv-elem x u I* $\Longrightarrow$ *intv-elem x u I′*
$\Longrightarrow$ $I = I'$
⟨*proof*⟩

lemma *valid-intv′-distinct*:
$-c \leq d \Longrightarrow$ *valid-intv′ c d I* $\Longrightarrow$ *valid-intv′ c d I′* $\Longrightarrow$ *intv′-elem x y u I*
$\Longrightarrow$ *intv′-elem x y u I′*
$\Longrightarrow$ $I = I'$
⟨*proof*⟩

From this we show that all valid regions are distinct

lemma *valid-regions-distinct*:

valid-region X k I J r $\Longrightarrow$ *valid-region X k I' J' r'* $\Longrightarrow$ $v \in$ *region X I J r* $\Longrightarrow$ $v \in$ *region X I' J' r'*
$\Longrightarrow$ *region X I J r* = *region X I' J' r'*
⟨*proof*⟩

locale *Beta-Regions* =
fixes *X k* $\mathcal{R}$ **and** *V* :: ('*c*, *t*) *cval set*
defines $\mathcal{R} \equiv \{$*region X I J r* | *I J r*. *valid-region X k I J r*$\}$
defines $V \equiv \{v\ .\ \forall\ x \in X.\ v\ x \geq 0\}$
assumes *finite*: *finite X*
assumes *non-empty*: $X \neq \{\}$
begin

lemma $\mathcal{R}$*-regions-distinct*:
$[\![R \in \mathcal{R};\ v \in R;\ R' \in \mathcal{R};\ R \neq R']\!] \Longrightarrow v \notin R'$
⟨*proof*⟩

Secondly, we also need to show that every valuations belongs to a region which is part of the partition.

definition *intv-of* :: *nat* ⇒ *t* ⇒ *intv* **where**
intv-of c v ≡
if (*v* > *c*) *then Greater c*
else if (∃ *x* :: *nat*. *x* = *v*) *then* (*Const* (*nat* (*floor v*)))
else (*Intv* (*nat* (*floor v*)))

definition *intv'-of* :: *int* ⇒ *int* ⇒ *t* ⇒ *intv'* **where**
intv'-of l u v ≡
if (*v* > *u*) *then Greater' u*
else if (*v* < *l*) *then Smaller' l*
else if (∃ *x* :: *int*. *x* = *v*) *then* (*Const'* (*floor v*))
else (*Intv'* (*floor v*))

lemma *region-cover*:
$\forall\ x \in X.\ v\ x \geq 0 \Longrightarrow \exists\ R.\ R \in \mathcal{R} \wedge v \in R$
⟨*proof*⟩

lemma *region-cover-V*: $v \in V \Longrightarrow \exists\ R.\ R \in \mathcal{R} \wedge v \in R$ ⟨*proof*⟩

Note that we cannot show that every region is non-empty anymore. The problem are regions fixing differences between an 'infeasible' constant.

We can show that there is always exactly one region a valid valuation belongs to. Note that we do not need non-emptiness for that.

lemma *regions-partition*:

$\forall\, x \in X.\ 0 \leq v\ x \Longrightarrow \exists!\ R \in \mathcal{R}.\ v \in R$
⟨*proof*⟩

lemma *region-unique*:
$v \in R \Longrightarrow R \in \mathcal{R} \Longrightarrow [v]_{\mathcal{R}} = R$
⟨*proof*⟩

lemma *regions-partition'*:
$\forall\, x{\in}X.\ 0 \leq v\ x \Longrightarrow \forall\, x{\in}X.\ 0 \leq v'\ x \Longrightarrow v' \in [v]_{\mathcal{R}} \Longrightarrow [v']_{\mathcal{R}} = [v]_{\mathcal{R}}$
⟨*proof*⟩

lemma *regions-closed*:
$R \in \mathcal{R} \Longrightarrow v \in R \Longrightarrow t \geq 0 \Longrightarrow [v \oplus t]_{\mathcal{R}} \in \mathcal{R}$
⟨*proof*⟩

lemma *regions-closed'*:
$R \in \mathcal{R} \Longrightarrow v \in R \Longrightarrow t \geq 0 \Longrightarrow (v \oplus t) \in [v \oplus t]_{\mathcal{R}}$
⟨*proof*⟩

lemma *valid-regions-I-cong*:
valid-region X k I J r $\Longrightarrow \forall\ x \in X.\ I\ x = I'\ x$
$\Longrightarrow \forall\ x \in X.\ \forall\ y \in X.\ (\mathit{isGreater}\ (I\ x) \vee \mathit{isGreater}\ (I\ y)) \longrightarrow J\ x\ y = J'\ x\ y$
$\Longrightarrow$ *region X I J r = region X I' J' r* $\wedge$ *valid-region X k I' J' r*
⟨*proof*⟩

fun *intv-const* :: *intv* $\Rightarrow$ *nat*
where
intv-const (*Const d*) = *d* |
intv-const (*Intv d*) = *d* |
intv-const (*Greater d*) = *d*

fun *intv'-const* :: *intv'* $\Rightarrow$ *int*
where
intv'-const (*Smaller' d*) = *d* |
intv'-const (*Const' d*) = *d* |
intv'-const (*Intv' d*) = *d* |
intv'-const (*Greater' d*) = *d*

lemma *finite-$\mathcal{R}$-aux*:
fixes *P A B* **assumes** *finite* {*x*. *A x*} *finite* {*x*. *B x*}
shows *finite* {(*I*, *J*) | *I J*. *P I J r* $\wedge$ *A I* $\wedge$ *B J*}
⟨*proof*⟩

lemma *finite-$\mathcal{R}$*:
notes [[*simproc add*: *finite-Collect*]]
shows *finite* $\mathcal{R}$
⟨*proof*⟩

end

6.3 Approximation with β-regions

locale *Beta-Regions'* = *Beta-Regions* +
fixes *v n not-in-X*
assumes *clock-numbering*: $\forall\ c.\ v\ c > 0 \wedge (\forall\, x.\ \forall\, y.\ v\ x \leq n \wedge v\ y \leq n \wedge v\ x = v\ y \longrightarrow x = y)$
$\forall\, k :: nat \leq n.\ k > 0 \longrightarrow (\exists\, c \in X.\ v\ c = k)\ \forall\ c \in X.\ v\ c \leq n$
assumes *not-in-X*: $not\text{-}in\text{-}X \notin X$
begin

definition $v' \equiv \lambda\ i.\ if\ i \leq n\ then\ (THE\ c.\ c \in X \wedge v\ c = i)\ else\ not\text{-}in\text{-}X$

lemma *v-v'*:
$\forall\ c \in X.\ v'\ (v\ c) = c$
⟨*proof*⟩

abbreviation
$vabstr\ (S :: ('a,\ t)\ zone)\ M \equiv S = [M]_{v,n} \wedge (\forall\ i{\leq}n.\ \forall\ j{\leq}n.\ M\ i\ j \neq \infty \longrightarrow get\text{-}const\ (M\ i\ j) \in \mathbb{Z})$

definition *normalized*:
$normalized\ M \equiv$
$(\forall\ i\ j.\ 0 < i \wedge i \leq n \wedge 0 < j \wedge j \leq n \wedge M\ i\ j \neq \infty \longrightarrow$
$Lt\ (-\ ((k\ o\ v')\ j)) \leq M\ i\ j \wedge M\ i\ j \leq Le\ ((k\ o\ v')\ i))$
$\wedge\ (\forall\ i \leq n.\ i > 0 \longrightarrow (M\ i\ 0 \leq Le\ ((k\ o\ v')\ i) \vee M\ i\ 0 = \infty) \wedge Lt\ (-\ ((k\ o\ v')\ i)) \leq M\ 0\ i)$

definition *apx-def*:
$Approx_\beta\ Z \equiv \bigcap\ \{S.\ \exists\ U\ M.\ S = \bigcup\ U \wedge U \subseteq \mathcal{R} \wedge Z \subseteq S \wedge vabstr\ S\ M \wedge normalized\ M\}$

lemma *apx-min*:
$S = \bigcup\ U \Longrightarrow U \subseteq \mathcal{R} \Longrightarrow S = [M]_{v,n} \Longrightarrow \forall\ i{\leq}n.\ \forall\ j{\leq}n.\ M\ i\ j \neq \infty \longrightarrow get\text{-}const\ (M\ i\ j) \in \mathbb{Z}$
$\Longrightarrow normalized\ M \Longrightarrow Z \subseteq S \Longrightarrow Approx_\beta\ Z \subseteq S$
⟨*proof*⟩

lemma $U \neq \{\} \Longrightarrow x \in \bigcap\ U \Longrightarrow \exists\ S \in U.\ x \in S$ ⟨*proof*⟩

lemma $\mathcal{R}$-*union*: $\bigcup\ \mathcal{R} = V$ ⟨*proof*⟩

lemma *all-dbm*: $\exists\ M.\ vabstr\ (\bigcup \mathcal{R})\ M \wedge normalized\ M$
⟨*proof*⟩

lemma $\mathcal{R}$-*int*:
$R \in \mathcal{R} \Longrightarrow R' \in \mathcal{R} \Longrightarrow R \neq R' \Longrightarrow R \cap R' = \{\}$ ⟨*proof*⟩

lemma *aux1*:
$u \in R \Longrightarrow R \in \mathcal{R} \Longrightarrow U \subseteq \mathcal{R} \Longrightarrow u \in \bigcup\ U \Longrightarrow R \subseteq \bigcup\ U$ ⟨*proof*⟩

lemma *aux2*: $x \in \bigcap\ U \Longrightarrow U \neq \{\} \Longrightarrow \exists\ S \in U.\ x \in S$ ⟨*proof*⟩

lemma *aux2′*: $x \in \bigcap\ U \Longrightarrow U \neq \{\} \Longrightarrow \forall\ S \in U.\ x \in S$ ⟨*proof*⟩

lemma *apx-subset*: $Z \subseteq Approx_\beta\ Z$ ⟨*proof*⟩

lemma *aux3*:
$\forall\ X \in U.\ \forall\ Y \in U.\ X \cap Y \in U \Longrightarrow S \subseteq U \Longrightarrow S \neq \{\} \Longrightarrow finite\ S$
$\Longrightarrow \bigcap\ S \in U$
⟨*proof*⟩

lemma *empty-zone-dbm*:
$\exists\ M :: t\ DBM.\ vabstr\ \{\}\ M \wedge normalized\ M \wedge (\forall\, k \leq n.\ M\ k\ k \leq Le\ 0)$
⟨*proof*⟩

lemma *valid-dbms-int*:
$\forall\, X{\in}\{S.\ \exists\, M.\ vabstr\ S\ M\}.\ \forall\, Y{\in}\{S.\ \exists\, M.\ vabstr\ S\ M\}.\ X \cap Y \in \{S.$
$\exists\, M.\ vabstr\ S\ M\}$
⟨*proof*⟩

print-statement *split-min*

lemma *split-min′*:
$P\ (min\ i\ j) = ((min\ i\ j = i \longrightarrow P\ i) \wedge (min\ i\ j = j \longrightarrow P\ j))$
⟨*proof*⟩

lemma *normalized-and-preservation*:
$normalized\ M1 \Longrightarrow normalized\ M2 \Longrightarrow normalized\ (And\ M1\ M2)$
⟨*proof*⟩

lemma *valid-dbms-int′*:
$\forall X{\in}\{S.\ \exists M.\ vabstr\ S\ M \wedge normalized\ M\}.\ \forall Y{\in}\{S.\ \exists M.\ vabstr\ S\ M \wedge normalized\ M\}.$
$X \cap Y \in \{S.\ \exists M.\ vabstr\ S\ M \wedge normalized\ M\}$
⟨*proof*⟩

lemma *apx-in*:
$Z \subseteq V \Longrightarrow Approx_\beta\ Z \in \{S.\ \exists\ U\ M.\ S = \bigcup\ U \wedge U \subseteq \mathcal{R} \wedge Z \subseteq S \wedge vabstr\ S\ M \wedge normalized\ M\}$
⟨*proof*⟩

lemma *apx-empty*:
$Approx_\beta\ \{\} = \{\}$
⟨*proof*⟩

end

6.4 Computing β-Approximation

context *Beta-Regions′*
begin

lemma *dbm-regions*:
$vabstr\ S\ M \Longrightarrow normalized\ M \Longrightarrow [M]_{v,n} \neq \{\} \Longrightarrow [M]_{v,n} \subseteq V \Longrightarrow \exists\ U \subseteq \mathcal{R}.\ S = \bigcup\ U$
⟨*proof*⟩

lemma *dbm-regions′*:
$vabstr\ S\ M \Longrightarrow normalized\ M \Longrightarrow S \subseteq V \Longrightarrow \exists\ U \subseteq \mathcal{R}.\ S = \bigcup\ U$
⟨*proof*⟩

lemma *dbm-regions″*:
$dbm\text{-}int\ M\ n \Longrightarrow normalized\ M \Longrightarrow [M]_{v,n} \subseteq V \Longrightarrow \exists\ U \subseteq \mathcal{R}.\ [M]_{v,n} = \bigcup\ U$
⟨*proof*⟩

lemma *canonical-saturated-1*:
assumes $Le\ r \leq M\ (v\ c1)\ 0$
and $Le\ (-\ r) \leq M\ 0\ (v\ c1)$
and *cycle-free M n*
and *canonical M n*
and $v\ c1 \leq n$
and $v\ c1 > 0$
and $\forall c.\ v\ c \leq n \longrightarrow 0 < v\ c$

obtains u **where** $u \in [M]_{v,n}\ u\ c1 = r$
⟨*proof*⟩

lemma *canonical-saturated-2*:
assumes $Le\ r \leq M\ 0\ (v\ c2)$
and $Le\ (-\ r) \leq M\ (v\ c2)\ 0$
and *cycle-free* $M\ n$
and *canonical* $M\ n$
and $v\ c2 \leq n$
and $v\ c2 > 0$
and $\forall\, c.\ v\ c \leq n \longrightarrow 0 < v\ c$
obtains u **where** $u \in [M]_{v,n}\ u\ c2 = -\ r$
⟨*proof*⟩

lemma *canonical-saturated-3*:
assumes $Le\ r \leq M\ (v\ c1)\ (v\ c2)$
and $Le\ (-\ r) \leq M\ (v\ c2)\ (v\ c1)$
and *cycle-free* $M\ n$
and *canonical* $M\ n$
and $v\ c1 \leq n\ v\ c2 \leq n$
and $v\ c1 \neq v\ c2$
and $\forall\, c.\ v\ c \leq n \longrightarrow 0 < v\ c$
obtains u **where** $u \in [M]_{v,n}\ u\ c1 - u\ c2 = r$
⟨*proof*⟩

lemma *DBM-canonical-subset-le*:
notes *any-le-inf*[*intro*]
fixes $M :: t\ DBM$
assumes *canonical* $M\ n\ [M]_{v,n} \subseteq [M']_{v,n}\ [M]_{v,n} \neq \{\}\ i \leq n\ j \leq n\ i \neq j$
shows $M\ i\ j \leq M'\ i\ j$
⟨*proof*⟩

lemma *DBM-set-diag*:
assumes $[M]_{v,n} \neq \{\}$
shows $[M]_{v,n} = [(\lambda\ i\ j.\ if\ i = j\ then\ Le\ 0\ else\ M\ i\ j)]_{v,n}$
⟨*proof*⟩

lemma *DBM-le-subset'*:
assumes $\forall\, i \leq n.\ \forall\ j \leq n.\ i \neq j \longrightarrow M\ i\ j \leq M'\ i\ j$
and $\forall\ i{\leq}n.\ M'\ i\ i \geq Le\ 0$
and $u \in [M]_{v,n}$
shows $u \in [M']_{v,n}$
⟨*proof*⟩

lemma *neg-diag-empty-spec*:
assumes $i \leq n$ M i $i < \mathbf{1}$
shows $[M]_{v,n} = \{\}$
⟨*proof*⟩

lemma *canonical-empty-zone-spec*:
assumes *canonical* M n
shows $[M]_{v,n} = \{\} \longleftrightarrow (\exists\, i{\leq}n.\ M\ i\ i < \mathbf{1})$
⟨*proof*⟩

lemma *norm-set-diag*:
assumes *canonical* M n $[M]_{v,n} \neq \{\}$
obtains M' **where** $[M]_{v,n} = [M']_{v,n}$ $[norm\ M\ (k\ o\ v')\ n]_{v,n} = [norm\ M'\ (k\ o\ v')\ n]_{v,n}$
$\forall\ i \leq n.\ M'\ i\ i = \mathbf{1}$ *canonical* M' n
⟨*proof*⟩

lemma *norm-normalizes*:
notes *any-le-inf*[*intro*]
shows *normalized* $(norm\ M\ (k\ o\ v')\ n)$
⟨*proof*⟩

lemma *norm-int-preservation*:
assumes *dbm-int* M n $i \leq n$ $j \leq n$ $norm\ M\ (k\ o\ v')\ n\ i\ j \neq \infty$
shows *get-const* $(norm\ M\ (k\ o\ v')\ n\ i\ j) \in \mathbb{Z}$
⟨*proof*⟩

lemma *norm-V-preservation'*:
notes *any-le-inf*[*intro*]
assumes $[M]_{v,n} \subseteq V$ *canonical* M n $[M]_{v,n} \neq \{\}$
shows $[norm\ M\ (k\ o\ v')\ n]_{v,n} \subseteq V$
⟨*proof*⟩

lemma *norm-V-preservation*:
assumes $[M]_{v,n} \subseteq V$ *canonical* M n
shows $[norm\ M\ (k\ o\ v')\ n]_{v,n} \subseteq V$ (**is** $[?M]_{v,n} \subseteq V$)
⟨*proof*⟩

lemma *norm-min*:
assumes *normalized* $M1$ $[M]_{v,n} \subseteq [M1]_{v,n}$
canonical M n $[M]_{v,n} \neq \{\}$ $[M]_{v,n} \subseteq V$
shows $[norm\ M\ (k\ o\ v')\ n]_{v,n} \subseteq [M1]_{v,n}$ (**is** $[?M2]_{v,n} \subseteq [M1]_{v,n}$)
⟨*proof*⟩

lemma *apx-norm-eq*:
assumes *canonical M n* $[M]_{v,n} \subseteq V$ *dbm-int M n*
shows $Approx_\beta\ ([M]_{v,n}) = [norm\ M\ (k\ o\ v')\ n]_{v,n}$
⟨*proof*⟩

end

6.5 Auxiliary β-boundedness Theorems

context *Beta-Regions'*
begin

lemma *β-boundedness-diag-lt*:
fixes $m :: int$
assumes $-k\ y \leq m\ m \leq k\ x\ x \in X\ y \in X$
shows $\exists\ U \subseteq \mathcal{R}.\ \bigcup\ U = \{u \in V.\ u\ x - u\ y < m\}$
⟨*proof*⟩

lemma *β-boundedness-diag-eq*:
fixes $m :: int$
assumes $-k\ y \leq m\ m \leq k\ x\ x \in X\ y \in X$
shows $\exists\ U \subseteq \mathcal{R}.\ \bigcup\ U = \{u \in V.\ u\ x - u\ y = m\}$
⟨*proof*⟩

lemma *β-boundedness-lt*:
fixes $m :: int$
assumes $m \leq k\ x\ x \in X$
shows $\exists\ U \subseteq \mathcal{R}.\ \bigcup\ U = \{u \in V.\ u\ x < m\}$
⟨*proof*⟩

lemma *β-boundedness-gt*:
fixes $m :: int$
assumes $m \leq k\ x\ x \in X$
shows $\exists\ U \subseteq \mathcal{R}.\ \bigcup\ U = \{u \in V.\ u\ x > m\}$
⟨*proof*⟩

lemma *β-boundedness-eq*:
fixes $m :: int$
assumes $m \leq k\ x\ x \in X$
shows $\exists\ U \subseteq \mathcal{R}.\ \bigcup\ U = \{u \in V.\ u\ x = m\}$
⟨*proof*⟩

lemma *β-boundedness-diag-le*:
fixes $m :: int$

assumes $-k\ y \leq m\ m \leq k\ x\ x \in X\ y \in X$
shows $\exists\ U \subseteq \mathcal{R}.\ \bigcup\ U = \{u \in V.\ u\ x - u\ y \leq m\}$
⟨*proof*⟩

lemma *β-boundedness-le*:
fixes $m :: int$
assumes $m \leq k\ x\ x \in X$
shows $\exists\ U \subseteq \mathcal{R}.\ \bigcup\ U = \{u \in V.\ u\ x \leq m\}$
⟨*proof*⟩

lemma *β-boundedness-ge*:
fixes $m :: int$
assumes $m \leq k\ x\ x \in X$
shows $\exists\ U \subseteq \mathcal{R}.\ \bigcup\ U = \{u \in V.\ u\ x \geq m\}$
⟨*proof*⟩

lemma *β-boundedness-diag-lt′*:
fixes $m :: int$
shows
$-k\ y \leq (m :: int) \Longrightarrow m \leq k\ x \Longrightarrow x \in X \Longrightarrow y \in X \Longrightarrow Z \subseteq \{u \in V.\ u\ x - u\ y < m\}$
$\Longrightarrow Approx_\beta\ Z \subseteq \{u \in V.\ u\ x - u\ y < m\}$
⟨*proof*⟩

lemma *β-boundedness-diag-le′*:
fixes $m :: int$
shows
$-k\ y \leq (m :: int) \Longrightarrow m \leq k\ x \Longrightarrow x \in X \Longrightarrow y \in X \Longrightarrow Z \subseteq \{u \in V.\ u\ x - u\ y \leq m\}$
$\Longrightarrow Approx_\beta\ Z \subseteq \{u \in V.\ u\ x - u\ y \leq m\}$
⟨*proof*⟩

lemma *β-boundedness-lt′*:
fixes $m :: int$
shows
$m \leq k\ x \Longrightarrow x \in X \Longrightarrow Z \subseteq \{u \in V.\ u\ x < m\} \Longrightarrow Approx_\beta\ Z \subseteq \{u \in V.\ u\ x < m\}$
⟨*proof*⟩

lemma *β-boundedness-gt′*:
fixes $m :: int$
shows
$m \leq k\ x \Longrightarrow x \in X \Longrightarrow Z \subseteq \{u \in V.\ u\ x > m\} \Longrightarrow Approx_\beta\ Z \subseteq \{u \in V.\ u\ x > m\}$

⟨*proof*⟩

lemma *obtains-dbm-le*:
fixes $m :: int$
assumes $x \in X\ m \leq k\ x$
obtains M **where** $vabstr\ \{u \in V.\ u\ x \leq m\}\ M\ normalized\ M$
⟨*proof*⟩

lemma β-*boundedness-le'*:
fixes $m :: int$
shows
$m \leq k\ x \Longrightarrow x \in X \Longrightarrow Z \subseteq \{u \in V.\ u\ x \leq m\} \Longrightarrow Approx_\beta\ Z \subseteq \{u \in V.\ u\ x \leq m\}$
⟨*proof*⟩

lemma *obtains-dbm-ge*:
fixes $m :: int$
assumes $x \in X\ m \leq k\ x$
obtains M **where** $vabstr\ \{u \in V.\ u\ x \geq m\}\ M\ normalized\ M$
⟨*proof*⟩

lemma β-*boundedness-ge'*:
fixes $m :: int$
shows $m \leq k\ x \Longrightarrow x \in X \Longrightarrow Z \subseteq \{u \in V.\ u\ x \geq m\} \Longrightarrow Approx_\beta\ Z \subseteq \{u \in V.\ u\ x \geq m\}$
⟨*proof*⟩

end

end

7 The Classic Construction for Decidability

theory *Regions*
imports *Timed-Automata Misc*
begin

The following is a formalization of regions in the correct version of Patricia Bouyer et al.

7.1 Definition of Regions

type-synonym $'c\ ceiling = ('c \Rightarrow nat)$

datatype *intv* =
  Const nat |
  Intv nat |
  Greater nat

type-synonym $t = real$

instantiation *real* :: *time*
begin
  instance ⟨*proof*⟩
end

inductive *valid-intv* :: *nat* ⇒ *intv* ⇒ *bool*
where
  $0 \leq d \Longrightarrow d \leq c \Longrightarrow$ *valid-intv c* (*Const d*) |
  $0 \leq d \Longrightarrow d < c \Longrightarrow$ *valid-intv c* (*Intv d*) |
  valid-intv c (*Greater c*)

inductive *intv-elem* :: *'c* ⇒ (*'c*,*t*) *cval* ⇒ *intv* ⇒ *bool*
where
  $u\ x = d \Longrightarrow$ *intv-elem x u* (*Const d*) |
  $d < u\ x \Longrightarrow u\ x < d + 1 \Longrightarrow$ *intv-elem x u* (*Intv d*) |
  $c < u\ x \Longrightarrow$ *intv-elem x u* (*Greater c*)

abbreviation *total-preorder r* ≡ *refl r* ∧ *trans r*

inductive *valid-region* :: *'c set* ⇒ (*'c* ⇒ *nat*) ⇒ (*'c* ⇒ *intv*) ⇒ *'c rel* ⇒ *bool*
where
  ⟦$X_0 = \{x \in X.\ \exists\ d.\ I\ x =$ *Intv d*$\}$; *refl-on* X_0 *r*; *trans r*; *total-on* X_0 *r*; $\forall\ x \in X.$ *valid-intv* (*k x*) (*I x*)⟧
  ⟹ *valid-region X k I r*

inductive-set *region* **for** *X I r*
where
  $\forall\ x \in X.\ u\ x \geq 0 \Longrightarrow \forall\ x \in X.$ *intv-elem x u* (*I x*) $\Longrightarrow X_0 = \{x \in X.\ \exists\ d.\ I\ x =$ *Intv d*$\} \Longrightarrow$
   $\forall\ x \in X_0.\ \forall\ y \in X_0.\ (x, y) \in r \longleftrightarrow$ *frac* (*u x*) ≤ *frac* (*u y*)
  ⟹ *u* ∈ *region X I r*

Defining the unique element of a partition that contains a valuation

definition *part* ([-]_ [61,61] 61) **where** *part v* $\mathcal{R} \equiv$ *THE R*. $R \in \mathcal{R} \wedge v \in R$

inductive-set *Succ* **for** $\mathcal{R}$ R **where**
$u \in R \Longrightarrow R \in \mathcal{R} \Longrightarrow R' \in \mathcal{R} \Longrightarrow t \geq 0 \Longrightarrow R' = [u \oplus t]_{\mathcal{R}} \Longrightarrow R' \in$ *Succ* $\mathcal{R}$ R

First we need to show that the set of regions is a partition of the set of all clock assignments. This property is only claimed by P. Bouyer.

inductive-cases[*elim*!]: *intv-elem x u* (*Const d*)
inductive-cases[*elim*!]: *intv-elem x u* (*Intv d*)
inductive-cases[*elim*!]: *intv-elem x u* (*Greater d*)
inductive-cases[*elim*!]: *valid-intv c* (*Greater d*)
inductive-cases[*elim*!]: *valid-intv c* (*Const d*)
inductive-cases[*elim*!]: *valid-intv c* (*Intv d*)

declare *valid-intv.intros*[*intro*]
declare *intv-elem.intros*[*intro*]
declare *Succ.intros*[*intro*]

declare *Succ.cases*[*elim*]

declare *region.cases*[*elim*]
declare *valid-region.cases*[*elim*]

7.2 Basic Properties

First we show that all valid intervals are distinct.

lemma *valid-intv-distinct*:
valid-intv c I $\Longrightarrow$ *valid-intv c I'* $\Longrightarrow$ *intv-elem x u I* $\Longrightarrow$ *intv-elem x u I'* $\Longrightarrow I = I'$
⟨*proof*⟩

From this we show that all valid regions are distinct.

lemma *valid-regions-distinct*:
valid-region X k I r $\Longrightarrow$ *valid-region X k I' r'* $\Longrightarrow v \in$ *region X I r* $\Longrightarrow v \in$ *region X I' r'*
$\Longrightarrow$ *region X I r* = *region X I' r'*
⟨*proof*⟩

lemma $\mathcal{R}$*-regions-distinct*:
$[\![\mathcal{R} = \{$*region X I r* $\mid$ *I r. valid-region X k I r*$\}$; $R \in \mathcal{R}$; $v \in R$; $R' \in \mathcal{R}$; $R \neq R']\!] \Longrightarrow v \notin R'$
⟨*proof*⟩

Secondly, we also need to show that every valuations belongs to a region which is part of the partition.

definition *intv-of* :: *nat* ⇒ *t* ⇒ *intv* **where**
intv-of k c ≡
if (*c* > *k*) *then Greater k*
else if (∃ *x* :: *nat*. *x* = *c*) *then* (*Const* (*nat* (*floor c*)))
else (*Intv* (*nat* (*floor c*)))

lemma *region-cover*:
∀ *x* ∈ *X*. *u x* ≥ *0* ⟹ ∃ *R*. *R* ∈ {*region X I r* | *I r*. *valid-region X k I r*}
∧ *u* ∈ *R*
⟨*proof*⟩

lemma *intv-not-empty*:
obtains *d* **where** *intv-elem x* (*v*(*x* := *d*)) (*I x*)
⟨*proof*⟩

fun *get-intv-val* :: *intv* ⇒ *real* ⇒ *real*
where
get-intv-val (*Const d*) - = *d* |
get-intv-val (*Intv d*) *f* = *d* + *f* |
get-intv-val (*Greater d*) - = *d* + *1*

lemma *region-not-empty-aux*:
assumes *0* < *f f* < *1 0* < *g g* < *1*
shows *frac* (*get-intv-val* (*Intv d*) *f*) ≤ *frac* (*get-intv-val* (*Intv d'*) *g*) ⟷
f ≤ *g*
⟨*proof*⟩

lemma *region-not-empty*:
assumes *finite X valid-region X k I r*
shows ∃ *u*. *u* ∈ *region X I r*
⟨*proof*⟩

Now we can show that there is always exactly one region a valid valuation belongs to.

lemma *regions-partition*:
$\mathcal{R}$ = {*region X I r* | *I r*. *valid-region X k I r*} ⟹ ∀ *x* ∈ *X*. *0* ≤ *u x* ⟹
∃! *R* ∈ $\mathcal{R}$. *u* ∈ *R*
⟨*proof*⟩

lemma *region-unique*:
$\mathcal{R}$ = {*region X I r* | *I r*. *valid-region X k I r*} ⟹ *u* ∈ *R* ⟹ *R* ∈ $\mathcal{R}$ ⟹

$[u]_{\mathcal{R}} = R$
⟨*proof*⟩

lemma *regions-partition'*:
$\mathcal{R} = \{region\ X\ I\ r \mid I\ r.\ valid\text{-}region\ X\ k\ I\ r\} \Longrightarrow \forall\, x{\in}X.\ 0 \leq v\ x \Longrightarrow \forall\, x{\in}X.\ 0 \leq v'\ x \Longrightarrow v' \in [v]_{\mathcal{R}}$
$\Longrightarrow [v']_{\mathcal{R}} = [v]_{\mathcal{R}}$
⟨*proof*⟩

lemma *regions-closed*:
$\mathcal{R} = \{region\ X\ I\ r \mid I\ r.\ valid\text{-}region\ X\ k\ I\ r\} \Longrightarrow R \in \mathcal{R} \Longrightarrow v \in R \Longrightarrow t \geq 0 \Longrightarrow [v \oplus t]_{\mathcal{R}} \in \mathcal{R}$
⟨*proof*⟩

lemma *regions-closed'*:
$\mathcal{R} = \{region\ X\ I\ r \mid I\ r.\ valid\text{-}region\ X\ k\ I\ r\} \Longrightarrow R \in \mathcal{R} \Longrightarrow v \in R \Longrightarrow t \geq 0 \Longrightarrow (v \oplus t) \in [v \oplus t]_{\mathcal{R}}$
⟨*proof*⟩

lemma *valid-regions-I-cong*:
$valid\text{-}region\ X\ k\ I\ r \Longrightarrow \forall\ x \in X.\ I\ x = I'\ x \Longrightarrow region\ X\ I\ r = region\ X\ I'\ r \wedge valid\text{-}region\ X\ k\ I'\ r$
⟨*proof*⟩

fun *intv-const* :: $intv \Rightarrow nat$
where
$intv\text{-}const\ (Const\ d) = d \mid$
$intv\text{-}const\ (Intv\ d) = d \mid$
$intv\text{-}const\ (Greater\ d) = d$

lemma *finite-$\mathcal{R}$*:
notes [[*simproc add*: *finite-Collect*]] *finite-subset*[*intro*]
fixes $X\ k$
defines $\mathcal{R} \equiv \{region\ X\ I\ r \mid I\ r.\ valid\text{-}region\ X\ k\ I\ r\}$
assumes *finite* X
shows *finite* $\mathcal{R}$
⟨*proof*⟩

lemma *SuccI2*:
$\mathcal{R} = \{region\ X\ I\ r \mid I\ r.\ valid\text{-}region\ X\ k\ I\ r\} \Longrightarrow v \in R \Longrightarrow R \in \mathcal{R} \Longrightarrow t \geq 0 \Longrightarrow R' = [v \oplus t]_{\mathcal{R}}$
$\Longrightarrow R' \in Succ\ \mathcal{R}\ R$
⟨*proof*⟩

7.3 Set of Regions

The first property Bouyer shows is that these regions form a 'set of regions'.

For the unbounded region in the upper right corner, the set of successors only contains itself.

lemma *Succ-refl*:
$\mathcal{R} = \{region\ X\ I\ r\ |I\ r.\ valid\text{-}region\ X\ k\ I\ r\} \Longrightarrow finite\ X \Longrightarrow R \in \mathcal{R} \Longrightarrow R \in Succ\ \mathcal{R}\ R$
⟨*proof*⟩

lemma *Succ-refl'*:
$\mathcal{R} = \{region\ X\ I\ r\ |I\ r.\ valid\text{-}region\ X\ k\ I\ r\} \Longrightarrow finite\ X \Longrightarrow \forall\ x \in X.\ \exists\ c.\ I\ x = Greater\ c$
$\Longrightarrow region\ X\ I\ r \in \mathcal{R} \Longrightarrow Succ\ \mathcal{R}\ (region\ X\ I\ r) = \{region\ X\ I\ r\}$
⟨*proof*⟩

Defining the closest successor of a region. Only exists if at least one interval is upper-bounded.

definition
$succ\ \mathcal{R}\ R =$
$(SOME\ R'.\ R' \in Succ\ \mathcal{R}\ R \wedge (\forall\ u \in R.\ \forall\ t \geq 0.\ (u \oplus t) \notin R \longrightarrow (\exists\ t' \leq t.\ (u \oplus t') \in R' \wedge 0 \leq t')))$

inductive *isConst* :: *intv* ⇒ *bool*
where
isConst (*Const* -)

inductive *isIntv* :: *intv* ⇒ *bool*
where
isIntv (*Intv* -)

inductive *isGreater* :: *intv* ⇒ *bool*
where
isGreater (*Greater* -)

declare *isIntv.intros*[*intro*!] *isConst.intros*[*intro*!] *isGreater.intros*[*intro*!]

declare *isIntv.cases*[*elim*!] *isConst.cases*[*elim*!] *isGreater.cases*[*elim*!]

What Bouyer states at the end. However, we have to be a bit more precise than in her statement.

lemma *closest-prestable-1*:
fixes *I X k r*

defines $\mathcal{R} \equiv \{region\ X\ I\ r\ |I\ r.\ valid\text{-}region\ X\ k\ I\ r\}$
defines $R \equiv region\ X\ I\ r$
defines $Z \equiv \{x \in X\ .\ \exists\ c.\ I\ x = Const\ c\}$
assumes $Z \neq \{\}$
defines $I' \equiv \lambda\ x.\ if\ x \notin Z\ then\ I\ x\ else\ if\ intv\text{-}const\ (I\ x) = k\ x\ then\ Greater\ (k\ x)\ else\ Intv\ (intv\text{-}const\ (I\ x))$
defines $r' \equiv r \cup \{(x,y)\ .\ x \in Z \wedge y \in X \wedge intv\text{-}const\ (I\ x) < k\ x \wedge isIntv\ (I'\ y)\}$
assumes *finite X*
assumes $valid\text{-}region\ X\ k\ I\ r$
shows $\forall\ v \in R.\ \forall\ t{>}0.\ \exists\, t'{\leq}t.\ (v \oplus t') \in region\ X\ I'\ r' \wedge t' \geq 0$
and $\forall\ v \in region\ X\ I'\ r'.\ \forall\ t{\geq}0.\ (v \oplus t) \notin R$
and $\forall\ x \in X.\ \neg\ isConst\ (I'\ x)$
and $\forall\ v \in R.\ \forall\ t < 1.\ \forall\ t' \geq 0.\ (v \oplus t') \in region\ X\ I'\ r'$
$\longrightarrow \{x.\ x \in X \wedge (\exists\ c.\ I\ x = Intv\ c \wedge v\ x + t \geq c + 1)\}$
$= \{x.\ x \in X \wedge (\exists\ c.\ I'\ x = Intv\ c \wedge (v \oplus t')\ x + (t - t') \geq c + 1)\}$
⟨*proof*⟩

lemma *closest-valid-1*:
fixes *I X k r*
defines $\mathcal{R} \equiv \{region\ X\ I\ r\ |I\ r.\ valid\text{-}region\ X\ k\ I\ r\}$
defines $R \equiv region\ X\ I\ r$
defines $Z \equiv \{x \in X\ .\ \exists\ c.\ I\ x = Const\ c\}$
assumes $Z \neq \{\}$
defines $I' \equiv \lambda\ x.\ if\ x \notin Z\ then\ I\ x\ else\ if\ intv\text{-}const\ (I\ x) = k\ x\ then\ Greater\ (k\ x)\ else\ Intv\ (intv\text{-}const\ (I\ x))$
defines $r' \equiv r \cup \{(x,y)\ .\ x \in Z \wedge y \in X \wedge intv\text{-}const\ (I\ x) < k\ x \wedge isIntv\ (I'\ y)\}$
assumes *finite X*
assumes $valid\text{-}region\ X\ k\ I\ r$
shows $valid\text{-}region\ X\ k\ I'\ r'$
⟨*proof*⟩

lemma *closest-prestable-2*:
fixes *I X k r*
defines $\mathcal{R} \equiv \{region\ X\ I\ r\ |I\ r.\ valid\text{-}region\ X\ k\ I\ r\}$
defines $R \equiv region\ X\ I\ r$
assumes $\forall\ x \in X.\ \neg\ isConst\ (I\ x)$
defines $X_0 \equiv \{x \in X.\ isIntv\ (I\ x)\}$
defines $M \equiv \{x \in X_0.\ \forall\ y \in X_0.\ (x,\ y) \in r \longrightarrow (y,\ x) \in r\}$
defines $I' \equiv \lambda\ x.\ if\ x \notin M\ then\ I\ x\ else\ Const\ (intv\text{-}const\ (I\ x) + 1)$
defines $r' \equiv \{(x,y) \in r.\ x \notin M \wedge y \notin M\}$
assumes *finite X*

assumes *valid-region X k I r*
assumes $M \neq \{\}$
shows $\forall\ v \in R.\ \forall\ t{\geq}0.\ (v \oplus t) \notin R \longrightarrow (\exists\, t'{\leq}t.\ (v \oplus t') \in \mathit{region}\ X\ I'\ r' \wedge t' \geq 0)$
and $\forall\ v \in \mathit{region}\ X\ I'\ r'.\ \forall\ t{\geq}0.\ (v \oplus t) \notin R$
and $\forall\ v \in R.\ \forall\ t'.\ \{x.\ x \in X \wedge (\exists\ c.\ I'\ x = \mathit{Intv}\ c \wedge (v \oplus t')\ x + (t - t') \geq \mathit{real}\ (c + 1))\}$
$= \{x.\ x \in X \wedge (\exists\ c.\ I\ x\ = \mathit{Intv}\ c \wedge v\ x + t \geq \mathit{real}\ (c + 1))\} - M$
and $\exists\ x \in X.\ \mathit{isConst}\ (I'\ x)$
⟨*proof*⟩

lemma *closest-valid-2*:
fixes *I X k r*
defines $\mathcal{R} \equiv \{\mathit{region}\ X\ I\ r\ |I\ r.\ \mathit{valid\text{-}region}\ X\ k\ I\ r\}$
defines $R \equiv \mathit{region}\ X\ I\ r$
assumes $\forall\ x \in X.\ \neg\ \mathit{isConst}\ (I\ x)$
defines $X_0 \equiv \{x \in X.\ \mathit{isIntv}\ (I\ x)\}$
defines $M \equiv \{x \in X_0.\ \forall\ y \in X_0.\ (x,\ y) \in r \longrightarrow (y,\ x) \in r\}$
defines $I' \equiv \lambda\ x.\ \mathit{if}\ x \notin M\ \mathit{then}\ I\ x\ \mathit{else}\ \mathit{Const}\ (\mathit{intv\text{-}const}\ (I\ x) + 1)$
defines $r' \equiv \{(x,y) \in r.\ x \notin M \wedge y \notin M\}$
assumes *finite X*
assumes *valid-region X k I r*
assumes $M \neq \{\}$
shows *valid-region X k I' r'*
⟨*proof*⟩

7.3.1 Putting the Proof for the 'Set of Regions' Property Together

Misc **lemma** *total-finite-trans-max*:
$X \neq \{\} \Longrightarrow \mathit{finite}\ X \Longrightarrow \mathit{total\text{-}on}\ X\ r \Longrightarrow \mathit{trans}\ r \Longrightarrow \exists\ x \in X.\ \forall\ y \in X.\ x \neq y \longrightarrow (y,\ x) \in r$
⟨*proof*⟩

lemma *card-mono-strict-subset*:
$\mathit{finite}\ A \Longrightarrow \mathit{finite}\ B \Longrightarrow \mathit{finite}\ C \Longrightarrow A \cap B \neq \{\} \Longrightarrow C = A - B \Longrightarrow \mathit{card}\ C < \mathit{card}\ A$
⟨*proof*⟩

Proof First we show that a shift by a non-negative integer constant means that any two valuations from the same region are being shifted to the same region.

lemma *int-shift-equiv*:
fixes X k **fixes** t :: *int*
defines $\mathcal{R} \equiv \{region\ X\ I\ r\ |I\ r.\ valid\text{-}region\ X\ k\ I\ r\}$
assumes $v \in R$ $v' \in R$ $R \in \mathcal{R}$ $t \geq 0$
shows $(v' \oplus t) \in [v \oplus t]_{\mathcal{R}}$ ⟨*proof*⟩

Now, we can use the 'immediate' induction proposed by P. Bouyer for shifts smaller than one. The induction principle is not at all obvious: the induction is over the set of clocks for which the valuation is shifted beyond the current interval boundaries. Using the two successor operations, we can see that either the set of these clocks remains the same (Z $=$) or strictly decreases (Z $=$).

lemma *set-of-regions-lt-1*:
fixes X k I r t v
defines $\mathcal{R} \equiv \{region\ X\ I\ r\ |I\ r.\ valid\text{-}region\ X\ k\ I\ r\}$
defines $C \equiv \{x.\ x \in X \wedge (\exists\ c.\ I\ x = Intv\ c \wedge v\ x + t \geq c + 1)\}$
assumes *valid-region* X k I r $v \in region\ X\ I\ r$ $v' \in region\ X\ I\ r$ *finite* X $0 \leq t$ $t < 1$
shows $\exists\ t' \geq 0.\ (v' \oplus t') \in [v \oplus t]_{\mathcal{R}}$ ⟨*proof*⟩

Finally, we can put the two pieces together: for a non-negative shift t, we first shift $\lfloor t \rfloor$ and then *frac* t.

lemma *set-of-regions*:
fixes X k
defines $\mathcal{R} \equiv \{region\ X\ I\ r\ |I\ r.\ valid\text{-}region\ X\ k\ I\ r\}$
assumes $R \in \mathcal{R}$ $v \in R$ $R' \in Succ\ \mathcal{R}\ R$ *finite* X
shows $\exists\ t \geq 0.\ [v \oplus t]_{\mathcal{R}} = R'$ ⟨*proof*⟩

7.4 Compability With Clock Constraints

definition *ccval* ({|-|} [*100*]) **where** $ccval\ cc \equiv \{v.\ v \vdash cc\}$

definition *ccompatible*
where
$ccompatible\ \mathcal{R}\ cc \equiv \forall\ R \in \mathcal{R}.\ R \subseteq ccval\ cc \vee ccval\ cc \cap R = \{\}$

lemma *ccompatible1*:
fixes X k **fixes** c :: *real*
defines $\mathcal{R} \equiv \{region\ X\ I\ r\ |I\ r.\ valid\text{-}region\ X\ k\ I\ r\}$
assumes $c \leq k$ x $c \in \mathbb{N}$ $x \in X$
shows $ccompatible\ \mathcal{R}\ (EQ\ x\ c)$ ⟨*proof*⟩

lemma *ccompatible2*:
fixes X k **fixes** c :: *real*

defines $\mathcal{R} \equiv \{region\ X\ I\ r\ |I\ r.\ valid\text{-}region\ X\ k\ I\ r\}$
assumes $c \leq k\ x\ c \in \mathbb{N}\ x \in X$
shows *ccompatible* $\mathcal{R}$ $(LT\ x\ c)$ ⟨*proof*⟩

lemma *ccompatible3*:
fixes $X\ k$ **fixes** c :: *real*
defines $\mathcal{R} \equiv \{region\ X\ I\ r\ |I\ r.\ valid\text{-}region\ X\ k\ I\ r\}$
assumes $c \leq k\ x\ c \in \mathbb{N}\ x \in X$
shows *ccompatible* $\mathcal{R}$ $(LE\ x\ c)$ ⟨*proof*⟩

lemma *ccompatible4*:
fixes $X\ k$ **fixes** c :: *real*
defines $\mathcal{R} \equiv \{region\ X\ I\ r\ |I\ r.\ valid\text{-}region\ X\ k\ I\ r\}$
assumes $c \leq k\ x\ c \in \mathbb{N}\ x \in X$
shows *ccompatible* $\mathcal{R}$ $(GT\ x\ c)$ ⟨*proof*⟩

lemma *ccompatible5*:
fixes $X\ k$ **fixes** c :: *real*
defines $\mathcal{R} \equiv \{region\ X\ I\ r\ |I\ r.\ valid\text{-}region\ X\ k\ I\ r\}$
assumes $c \leq k\ x\ c \in \mathbb{N}\ x \in X$
shows *ccompatible* $\mathcal{R}$ $(GE\ x\ c)$ ⟨*proof*⟩

lemma *ccompatible*:
fixes $X\ k$ **fixes** c :: *nat*
defines $\mathcal{R} \equiv \{region\ X\ I\ r\ |I\ r.\ valid\text{-}region\ X\ k\ I\ r\}$
assumes $\forall (x,m) \in collect\text{-}clock\text{-}pairs\ cc.\ m \leq k\ x \wedge x \in X \wedge m \in \mathbb{N}$
shows *ccompatible* $\mathcal{R}$ *cc* ⟨*proof*⟩

7.5 Compability with Resets

definition *region-set*
where
$region\text{-}set\ R\ x\ c = \{v(x := c)\ |\ v.\ v \in R\}$

lemma *region-set-id*:
fixes $X\ k$
defines $\mathcal{R} \equiv \{region\ X\ I\ r\ |I\ r.\ valid\text{-}region\ X\ k\ I\ r\}$
assumes $R \in \mathcal{R}\ v \in R\ finite\ X\ 0 \leq c\ c \leq k\ x\ x \in X$
shows $[v(x := c)]_{\mathcal{R}} = region\text{-}set\ R\ x\ c\ [v(x := c)]_{\mathcal{R}} \in \mathcal{R}\ v(x := c) \in [v(x := c)]_{\mathcal{R}}$
⟨*proof*⟩

definition *region-set′*
where

region-set′ R r c = $\{[r \to c]v \mid v.\ v \in R\}$

lemma *region-set′-id*:
fixes *X k* **and** *c* :: *nat*
defines $\mathcal{R} \equiv \{region\ X\ I\ r \mid I\ r.\ valid\text{-}region\ X\ k\ I\ r\}$
assumes $R \in \mathcal{R}$ $v \in R$ *finite X 0* $\leq c \forall\ x \in set\ r.\ c \leq k\ x\ set\ r \subseteq X$
shows $[[r \to c]v]_{\mathcal{R}} = region\text{-}set'\ R\ r\ c \wedge [[r \to c]v]_{\mathcal{R}} \in \mathcal{R} \wedge [r \to c]v \in [[r \to c]v]_{\mathcal{R}}$ ⟨*proof*⟩

7.6 A Semantics Based on Regions

7.6.1 Single step

inductive *step-r* ::
('*a*, '*c*, *t*, '*s*) *ta* ⇒ ('*c*, *t*) *zone set* ⇒ '*s* ⇒ ('*c*, *t*) *zone* ⇒ '*s* ⇒ ('*c*, *t*) *zone* ⇒ *bool*
(-,- ⊢ ⟨-, -⟩ ⇝ ⟨-, -⟩ [*61*,*61*,*61*,*61*] *61*)
where
step-t-r:
$[\![\mathcal{R} = \{region\ X\ I\ r \mid I\ r.\ valid\text{-}region\ X\ k\ I\ r\};\ valid\text{-}abstraction\ A\ X\ k;\ R \in \mathcal{R};\ R' \in Succ\ \mathcal{R}\ R;$
$R \subseteq \{\!|inv\text{-}of\ A\ l|\!\};\ R' \subseteq \{\!|inv\text{-}of\ A\ l|\!\}]\!] \Longrightarrow A,\mathcal{R} \vdash \langle l,R\rangle \rightsquigarrow \langle l,R'\rangle \mid$
step-a-r:
$[\![\mathcal{R} = \{region\ X\ I\ r \mid I\ r.\ valid\text{-}region\ X\ k\ I\ r\};\ valid\text{-}abstraction\ A\ X\ k;\ A \vdash l \longrightarrow^{g,a,r} l';\ R \in \mathcal{R}]\!]$
$\Longrightarrow A,\mathcal{R} \vdash \langle l,R\rangle \rightsquigarrow \langle l',region\text{-}set'\ (R \cap \{u.\ u \vdash g\})\ r\ 0 \cap \{u.\ u \vdash inv\text{-}of\ A\ l'\}\rangle$

inductive-cases[*elim*!]: $A,\mathcal{R} \vdash \langle l,\ u\rangle \rightsquigarrow \langle l',\ u'\rangle$

declare *step-r.intros*[*intro*]

lemma *region-cover′*:
assumes $\mathcal{R} = \{region\ X\ I\ r \mid I\ r.\ valid\text{-}region\ X\ k\ I\ r\}$ **and** $\forall x{\in}X.\ 0 \leq v\ x$
shows $v \in [v]_{\mathcal{R}}$ $[v]_{\mathcal{R}} \in \mathcal{R}$
⟨*proof*⟩

lemma *step-r-complete-aux*:
fixes *R r A l′ g*
defines $R' \equiv region\text{-}set'\ (R \cap \{u.\ u \vdash g\})\ r\ 0 \cap \{u.\ u \vdash inv\text{-}of\ A\ l'\}$
assumes $\mathcal{R} = \{region\ X\ I\ r \mid I\ r.\ valid\text{-}region\ X\ k\ I\ r\}$
and *valid-abstraction A X k*
and $u \in R$

and $R \in \mathcal{R}$
and $A \vdash l \longrightarrow^{g,a,r} l'$
and $u \vdash g$
and $[r \rightarrow 0]u \vdash \mathit{inv\text{-}of}\ A\ l'$
shows $R = R \cap \{u.\ u \vdash g\} \wedge R' = \mathit{region\text{-}set'}\ R\ r\ 0 \wedge R' \in \mathcal{R}$
⟨*proof*⟩

lemma *step-r-complete*:
$[\![A \vdash \langle l,\ u\rangle \rightarrow \langle l',u'\rangle;\ \mathcal{R} = \{\mathit{region}\ X\ I\ r\ |I\ r.\ \mathit{valid\text{-}region}\ X\ k\ I\ r\};\ \mathit{valid\text{-}abstraction}\ A\ X\ k;$
$\forall\ x \in X.\ u\ x \geq 0]\!] \Longrightarrow \exists\ R'.\ A,\mathcal{R} \vdash \langle l,\ ([u]_{\mathcal{R}})\rangle \rightsquigarrow \langle l',R'\rangle \wedge u' \in R' \wedge R' \in \mathcal{R}$
⟨*proof*⟩

Compare this to lemma *step-z-sound*. This version is weaker because for regions we may very well arrive at a successor for which not every valuation can be reached by the predecessor. This is the case for e.g. the region with only Greater (k x) bounds.

lemma *step-r-sound*:
$A,\mathcal{R} \vdash \langle l,\ R\rangle \rightsquigarrow \langle l',R'\rangle \Longrightarrow \mathcal{R} = \{\mathit{region}\ X\ I\ r\ |I\ r.\ \mathit{valid\text{-}region}\ X\ k\ I\ r\}$
$\Longrightarrow R' \neq \{\} \Longrightarrow (\forall\ u \in R.\ \exists\ u' \in R'.\ A \vdash \langle l,\ u\rangle \rightarrow \langle l',u'\rangle)$
⟨*proof*⟩

7.6.2 Multi Step

inductive
steps-r :: $('a,\ 'c,\ t,\ 's)\ ta \Rightarrow ('c,\ t)\ \mathit{zone\ set} \Rightarrow 's \Rightarrow ('c,\ t)\ \mathit{zone} \Rightarrow 's \Rightarrow ('c,\ t)\ \mathit{zone} \Rightarrow \mathit{bool}$
(-,- ⊢ ⟨-, -⟩ ⇝* ⟨-, -⟩ [61,61,61,61,61,61] 61)
where
refl: $A,\mathcal{R} \vdash \langle l,\ R\rangle \rightsquigarrow* \langle l,\ R\rangle$ |
step: $A,\mathcal{R} \vdash \langle l,\ R\rangle \rightsquigarrow* \langle l',\ R'\rangle \Longrightarrow A,\mathcal{R} \vdash \langle l',\ R'\rangle \rightsquigarrow \langle l'',\ R''\rangle \Longrightarrow A,\mathcal{R} \vdash \langle l,\ R\rangle \rightsquigarrow* \langle l'',\ R''\rangle$

declare *steps-r.intros*[*intro*]

lemma *steps-alt*:
$A \vdash \langle l,\ u\rangle \rightarrow* \langle l',u'\rangle \Longrightarrow A \vdash \langle l',\ u'\rangle \rightarrow \langle l'',u''\rangle \Longrightarrow A \vdash \langle l,\ u\rangle \rightarrow* \langle l'',u''\rangle$
⟨*proof*⟩

lemma *emptiness-preservance*: $A,\mathcal{R} \vdash \langle l,\ R\rangle \rightsquigarrow \langle l',R'\rangle \Longrightarrow R = \{\} \Longrightarrow R' = \{\}$
⟨*proof*⟩

lemma *emptiness-preservance-steps*: $A,\mathcal{R} \vdash \langle l,\ R\rangle \rightsquigarrow* \langle l',R'\rangle \Longrightarrow R = \{\} \Longrightarrow R' = \{\}$
⟨*proof*⟩

Note how it is important to define the multi-step semantics "the right way round". This also the direction Bouyer implies for her implicit induction.

lemma *steps-r-sound*:
$A,\mathcal{R} \vdash \langle l,\ R\rangle \rightsquigarrow* \langle l',\ R'\rangle \Longrightarrow \mathcal{R} = \{region\ X\ I\ r\ |I\ r.\ valid\text{-}region\ X\ k\ I\ r\}$
$\Longrightarrow R' \neq \{\} \Longrightarrow u \in R \Longrightarrow \exists\ u' \in R'.\ A \vdash \langle l,\ u\rangle \rightarrow* \langle l',\ u'\rangle$
⟨*proof*⟩

lemma *steps-r-sound'*:
$A,\mathcal{R} \vdash \langle l,\ R\rangle \rightsquigarrow* \langle l',\ R'\rangle \Longrightarrow \mathcal{R} = \{region\ X\ I\ r\ |I\ r.\ valid\text{-}region\ X\ k\ I\ r\}$
$\Longrightarrow R' \neq \{\} \Longrightarrow (\exists\ u' \in R'.\ \exists\ u \in R.\ \ A \vdash \langle l,\ u\rangle \rightarrow* \langle l',\ u'\rangle)$
⟨*proof*⟩

lemma *single-step-r*:
$A,\mathcal{R} \vdash \langle l,\ R\rangle \rightsquigarrow \langle l',\ R'\rangle \Longrightarrow A,\mathcal{R} \vdash \langle l,\ R\rangle \rightsquigarrow* \langle l',\ R'\rangle$
⟨*proof*⟩

lemma *steps-r-alt*:
$A,\mathcal{R} \vdash \langle l',\ R'\rangle \rightsquigarrow* \langle l'',\ R''\rangle \Longrightarrow A,\mathcal{R} \vdash \langle l,\ R\rangle \rightsquigarrow \langle l',\ R'\rangle \Longrightarrow A,\mathcal{R} \vdash \langle l,\ R\rangle \rightsquigarrow* \langle l'',\ R''\rangle$
⟨*proof*⟩

lemma *single-step*:
$x1 \vdash \langle x2,\ x3\rangle \rightarrow \langle x4,x5\rangle \Longrightarrow x1 \vdash \langle x2,\ x3\rangle \rightarrow* \langle x4,x5\rangle$
⟨*proof*⟩

lemma *steps-r-complete*:
$[\![A \vdash \langle l,\ u\rangle \rightarrow* \langle l',u'\rangle;\ \mathcal{R} = \{region\ X\ I\ r\ |I\ r.\ valid\text{-}region\ X\ k\ I\ r\};$
$valid\text{-}abstraction\ A\ X\ k;$
$\forall\ x \in X.\ u\ x \geq 0]\!] \Longrightarrow \exists\ R'.\ A,\mathcal{R} \vdash \langle l,\ ([u]_{\mathcal{R}})\rangle \rightsquigarrow* \langle l',R'\rangle \wedge u' \in R'$
⟨*proof*⟩

end
theory *Closure*
imports *Regions*
begin

7.7 Correct Approximation of Zones with α-regions

locale *AlphaClosure* =
  fixes X k $\mathcal{R}$ **and** $V :: ('c, t)$ *cval set*
  defines $\mathcal{R} \equiv \{region\ X\ I\ r \mid I\ r.\ valid\text{-}region\ X\ k\ I\ r\}$
  defines $V \equiv \{v\ .\ \forall\ x \in X.\ v\ x \geq 0\}$
  assumes *finite*: *finite* X
begin

lemmas *set-of-regions-spec* = *set-of-regions*[*OF - - - finite*, *of - k*, *folded* $\mathcal{R}$*-def*]
lemmas *region-cover-spec* = *region-cover*[*of X - k*, *folded* $\mathcal{R}$*-def*]
lemmas *region-unique-spec* = *region-unique*[*of* $\mathcal{R}$ *X k*, *folded* $\mathcal{R}$*-def*, *simplified*]
lemmas *regions-closed'-spec* = *regions-closed'*[*of* $\mathcal{R}$ *X k*, *folded* $\mathcal{R}$*-def*, *simplified*]

lemma *valid-regions-distinct-spec*:
  $R \in \mathcal{R} \Longrightarrow R' \in \mathcal{R} \Longrightarrow v \in R \Longrightarrow v \in R' \Longrightarrow R = R'$
⟨*proof*⟩

definition *cla* ($Closure_\alpha$ - [*71*] *71*)
where
  $cla\ Z = \bigcup\ \{R \in \mathcal{R}.\ R \cap Z \neq \{\}\}$

The nice and easy properties proved by Bouyer **lemma** *closure-constraint-id*:
  $\forall (x,\ m) \in collect\text{-}clock\text{-}pairs\ g.\ m \leq real\ (k\ x) \wedge x \in X \wedge m \in \mathbb{N} \Longrightarrow Closure_\alpha\ \{\!|g|\!\} = \{\!|g|\!\} \cap V$
⟨*proof*⟩

lemma *closure-id'*:
  $Z \neq \{\} \Longrightarrow Z \subseteq R \Longrightarrow R \in \mathcal{R} \Longrightarrow Closure_\alpha\ Z = R$
⟨*proof*⟩

lemma *closure-id*:
  $Closure_\alpha\ Z \neq \{\} \Longrightarrow Z \subseteq R \Longrightarrow R \in \mathcal{R} \Longrightarrow Closure_\alpha\ Z = R$
⟨*proof*⟩

lemma *closure-update-mono*:
  $Z \subseteq V \Longrightarrow set\ r \subseteq X \Longrightarrow zone\text{-}set\ (Closure_\alpha\ Z)\ r \subseteq Closure_\alpha(zone\text{-}set\ Z\ r)$
⟨*proof*⟩

lemma *SuccI3*:

$R \in \mathcal{R} \Longrightarrow v \in R \Longrightarrow t \geq 0 \Longrightarrow (v \oplus t) \in R' \Longrightarrow R' \in \mathcal{R} \Longrightarrow R' \in Succ\ \mathcal{R}\ R$
⟨*proof*⟩

lemma *closure-delay-mono*:
$Z \subseteq V \Longrightarrow (Closure_\alpha\ Z)^\uparrow \subseteq Closure_\alpha\ (Z^\uparrow)$
⟨*proof*⟩

lemma *region-V*: $R \in \mathcal{R} \Longrightarrow R \subseteq V$ ⟨*proof*⟩

lemma *closure-V*:
$Closure_\alpha\ Z \subseteq V$
⟨*proof*⟩

lemma *closure-V-int*:
$Closure_\alpha\ Z = Closure_\alpha\ (Z \cap V)$
⟨*proof*⟩

lemma *closure-constraint-mono*:
$Closure_\alpha\ g = g \Longrightarrow g \cap (Closure_\alpha\ Z) \subseteq Closure_\alpha\ (g \cap Z)$
⟨*proof*⟩

lemma *closure-constraint-mono′*:
assumes $Closure_\alpha\ g = g \cap V$
shows $g \cap (Closure_\alpha\ Z) \subseteq Closure_\alpha\ (g \cap Z)$
⟨*proof*⟩

lemma *cla-empty-iff*:
$Z \subseteq V \Longrightarrow Z = \{\} \longleftrightarrow Closure_\alpha\ Z = \{\}$
⟨*proof*⟩

lemma *closure-involutive-aux*:
$U \subseteq \mathcal{R} \Longrightarrow Closure_\alpha \bigcup U = \bigcup U$
⟨*proof*⟩

lemma *closure-involutive-aux′*:
$\exists\ U.\ U \subseteq \mathcal{R} \wedge Closure_\alpha\ Z = \bigcup U$
⟨*proof*⟩

lemma *closure-involutive*:
$Closure_\alpha\ Closure_\alpha\ Z = Closure_\alpha\ Z$
⟨*proof*⟩

lemma *closure-involutive′*:

$Z \subseteq Closure_\alpha\ W \Longrightarrow Closure_\alpha\ Z \subseteq Closure_\alpha\ W$
⟨*proof*⟩

lemma *closure-subs*:
$Z \subseteq V \Longrightarrow Z \subseteq Closure_\alpha\ Z$
⟨*proof*⟩

lemma *cla-mono′*:
$Z' \subseteq V \Longrightarrow Z \subseteq Z' \Longrightarrow Closure_\alpha\ Z \subseteq Closure_\alpha\ Z'$
⟨*proof*⟩

lemma *cla-mono*:
$Z \subseteq Z' \Longrightarrow Closure_\alpha\ Z \subseteq Closure_\alpha\ Z'$
⟨*proof*⟩

7.8 A New Zone Semantics Abstracting with $Closure_\alpha$

7.8.1 Single step

inductive *step-z-alpha* ::
$('a, 'c, t, 's)\ ta \Rightarrow 's \Rightarrow ('c, t)\ zone \Rightarrow 's \Rightarrow ('c, t)\ zone \Rightarrow bool$
$(\text{-} \vdash \langle \text{-}, \text{-} \rangle \rightsquigarrow_\alpha \langle \text{-}, \text{-} \rangle\ [61,61,61]\ 61)$
where
step-alpha: $A \vdash \langle l, Z \rangle \rightsquigarrow \langle l', Z' \rangle \Longrightarrow A \vdash \langle l, Z \rangle \rightsquigarrow_\alpha \langle l', Closure_\alpha\ Z' \rangle$

inductive-cases[*elim*!]: $A \vdash \langle l, u \rangle \rightsquigarrow_\alpha \langle l', u' \rangle$

declare *step-z-alpha.intros*[*intro*]

lemma *up-V*: $Z \subseteq V \Longrightarrow Z^\uparrow \subseteq V$
⟨*proof*⟩

lemma *reset-V*: $Z \subseteq V \Longrightarrow (zone\text{-}set\ Z\ r) \subseteq V$
⟨*proof*⟩

lemma *step-z-V*: $A \vdash \langle l, Z \rangle \rightsquigarrow \langle l', Z' \rangle \Longrightarrow Z \subseteq V \Longrightarrow Z' \subseteq V$
⟨*proof*⟩

Single-step soundness and completeness follows trivially from *cla-empty-iff*.

lemma *step-z-alpha-sound*:
$A \vdash \langle l, Z \rangle \rightsquigarrow_\alpha \langle l', Z' \rangle \Longrightarrow Z \subseteq V \Longrightarrow Z' \neq \{\} \Longrightarrow \exists\ Z''.\ A \vdash \langle l, Z \rangle \rightsquigarrow \langle l', Z'' \rangle \wedge Z'' \neq \{\}$
⟨*proof*⟩

lemma *step-z-alpha-complete*:

$A \vdash \langle l,\ Z\rangle \rightsquigarrow \langle l',Z'\rangle \Longrightarrow Z \subseteq V \Longrightarrow Z' \neq \{\} \Longrightarrow \exists\ Z''.\ A \vdash \langle l,\ Z\rangle \rightsquigarrow_\alpha \langle l',Z''\rangle \wedge Z'' \neq \{\}$
⟨*proof*⟩

lemma *zone-set-mono*:
$A \subseteq B \Longrightarrow$ *zone-set* $A\ r \subseteq$ *zone-set* $B\ r$
⟨*proof*⟩

lemma *zone-delay-mono*:
$A \subseteq B \Longrightarrow A^{\uparrow} \subseteq B^{\uparrow}$
⟨*proof*⟩

7.8.2 Multi step

inductive
steps-z-alpha :: $('a,\ 'c,\ t,\ 's)\ ta \Rightarrow 's \Rightarrow ('c,\ t)\ zone \Rightarrow 's \Rightarrow ('c,\ t)\ zone \Rightarrow bool$
(- ⊢ ⟨-, -⟩ $\rightsquigarrow_\alpha$* ⟨-, -⟩ [*61*,*61*,*61*] *61*)
where
refl: $A \vdash \langle l,\ Z\rangle \rightsquigarrow_\alpha{*}\ \langle l,\ Z\rangle$ |
step: $A \vdash \langle l,\ Z\rangle \rightsquigarrow_\alpha{*}\ \langle l',\ Z'\rangle \Longrightarrow A \vdash \langle l',\ Z'\rangle \rightsquigarrow_\alpha \langle l'',\ Z''\rangle \Longrightarrow A \vdash \langle l,\ Z\rangle \rightsquigarrow_\alpha{*}\ \langle l'',\ Z''\rangle$

declare *steps-z-alpha.intros*[*intro*]

lemma *subset-int-mono*: $A \subseteq B \Longrightarrow A \cap C \subseteq B \cap C$ ⟨*proof*⟩

P. Bouyer's calculation for $Post\ (Closure_\alpha\ Z,\ e) \subseteq Closure_\alpha\ Post\ (Z,\ e)$

This is now obsolete as we argue solely with monotonicty of *steps-z* w.r.t $Closure_\alpha$

lemma *calc*:
valid-abstraction $A\ X\ k \Longrightarrow Z \subseteq V \Longrightarrow A \vdash \langle l,\ Closure_\alpha\ Z\rangle \rightsquigarrow \langle l',\ Z'\rangle$
$\Longrightarrow \exists Z''.\ A \vdash \langle l,\ Z\rangle \rightsquigarrow_\alpha \langle l',\ Z''\rangle \wedge Z' \subseteq Z''$
⟨*proof*⟩

Turning P. Bouyers argument for multiple steps into an inductive proof is not direct. With this initial argument we can get to a point where the induction hypothesis is applicable. This breaks the "information hiding" induced by the different variants of steps.

lemma *steps-z-alpha-closure-involutive'-aux*:
$A \vdash \langle l,\ Z\rangle \rightsquigarrow \langle l',Z'\rangle \Longrightarrow Closure_\alpha\ Z \subseteq Closure_\alpha\ W \Longrightarrow$ *valid-abstraction* $A\ X\ k \Longrightarrow Z \subseteq V$
$\Longrightarrow \exists\ W'.\ A \vdash \langle l,\ W\rangle \rightsquigarrow \langle l',W'\rangle \wedge Closure_\alpha\ Z' \subseteq Closure_\alpha\ W'$
⟨*proof*⟩

lemma *steps-z-alpha-closure-involutive'-aux'*:
$A \vdash \langle l,\ Z\rangle \rightsquigarrow \langle l',Z'\rangle \Longrightarrow \mathit{Closure}_\alpha\ Z \subseteq \mathit{Closure}_\alpha\ W \Longrightarrow \mathit{valid\text{-}abstraction}\ A\ X\ k \Longrightarrow Z \subseteq V \Longrightarrow W \subseteq Z$
$\Longrightarrow \exists\ W'.\ A \vdash \langle l,\ W\rangle \rightsquigarrow \langle l',W'\rangle \wedge \mathit{Closure}_\alpha\ Z' \subseteq \mathit{Closure}_\alpha\ W' \wedge W' \subseteq Z'$
⟨*proof*⟩

lemma *steps-z-alt*:
$A \vdash \langle l,\ Z\rangle \rightsquigarrow* \langle l',Z'\rangle \Longrightarrow A \vdash \langle l',\ Z'\rangle \rightsquigarrow \langle l'',Z''\rangle \Longrightarrow A \vdash \langle l,\ Z\rangle \rightsquigarrow* \langle l'',Z''\rangle$
⟨*proof*⟩

lemma *steps-z-alpha-V*: $A \vdash \langle l,\ Z\rangle \rightsquigarrow_\alpha* \langle l',Z'\rangle \Longrightarrow Z \subseteq V \Longrightarrow Z' \subseteq V$
⟨*proof*⟩

lemma *steps-z-alpha-closure-involutive'*:
$A \vdash \langle l,\ Z\rangle \rightsquigarrow_\alpha* \langle l',Z'\rangle \Longrightarrow A \vdash \langle l',\ Z'\rangle \rightsquigarrow \langle l'',Z''\rangle \Longrightarrow \mathit{valid\text{-}abstraction}\ A\ X\ k \Longrightarrow Z \subseteq V$
$\Longrightarrow \exists\ Z'''.\ A \vdash \langle l,\ Z\rangle \rightsquigarrow* \langle l'',Z'''\rangle \wedge \mathit{Closure}_\alpha\ Z'' \subseteq \mathit{Closure}_\alpha\ Z''' \wedge Z''' \subseteq Z''$
⟨*proof*⟩

Old proof using Bouyer's calculation

lemma *steps-z-alpha-closure-involutive''*:
$A \vdash \langle l,\ Z\rangle \rightsquigarrow_\alpha* \langle l',Z'\rangle \Longrightarrow A \vdash \langle l',\ Z'\rangle \rightsquigarrow \langle l'',Z''\rangle \Longrightarrow \mathit{valid\text{-}abstraction}\ A\ X\ k \Longrightarrow Z \subseteq V$
$\Longrightarrow \exists\ Z'''.\ A \vdash \langle l,\ Z\rangle \rightsquigarrow* \langle l'',Z'''\rangle \wedge \mathit{Closure}_\alpha\ Z'' \subseteq \mathit{Closure}_\alpha\ Z'''$
⟨*proof*⟩

lemma *steps-z-alpha-closure-involutive*:
$A \vdash \langle l,\ Z\rangle \rightsquigarrow_\alpha* \langle l',Z'\rangle \Longrightarrow \mathit{valid\text{-}abstraction}\ A\ X\ k \Longrightarrow Z \subseteq V$
$\Longrightarrow \exists\ Z''.\ A \vdash \langle l,\ Z\rangle \rightsquigarrow* \langle l',Z''\rangle \wedge \mathit{Closure}_\alpha\ Z' \subseteq \mathit{Closure}_\alpha\ Z'' \wedge Z'' \subseteq Z'$
⟨*proof*⟩

lemma *steps-z-V*:
$A \vdash \langle l,\ Z\rangle \rightsquigarrow* \langle l',Z'\rangle \Longrightarrow Z \subseteq V \Longrightarrow Z' \subseteq V$
⟨*proof*⟩

lemma *steps-z-alpha-sound*:
$A \vdash \langle l,\ Z\rangle \rightsquigarrow_\alpha* \langle l',Z'\rangle \Longrightarrow \mathit{valid\text{-}abstraction}\ A\ X\ k \Longrightarrow Z \subseteq V \Longrightarrow Z' \neq \{\}$
$\Longrightarrow \exists\ Z''.\ A \vdash \langle l,\ Z\rangle \rightsquigarrow* \langle l',Z''\rangle \wedge Z'' \neq \{\} \wedge Z'' \subseteq Z'$

⟨*proof*⟩

lemma *step-z-mono*:
$A \vdash \langle l,\ Z\rangle \rightsquigarrow \langle l',Z'\rangle \Longrightarrow Z \subseteq W \Longrightarrow \exists\ W'.\ A \vdash \langle l,\ W\rangle \rightsquigarrow \langle l',W'\rangle \wedge Z' \subseteq W'$
⟨*proof*⟩

lemma *step-z-alpha-mono*:
$A \vdash \langle l,\ Z\rangle \rightsquigarrow_\alpha \langle l',Z'\rangle \Longrightarrow Z \subseteq W \Longrightarrow W \subseteq V \Longrightarrow \exists\ W'.\ A \vdash \langle l,\ W\rangle \rightsquigarrow_\alpha \langle l',W'\rangle \wedge Z' \subseteq W'$
⟨*proof*⟩

lemma *steps-z-alpha-mono*:
$A \vdash \langle l,\ Z\rangle \rightsquigarrow_\alpha* \langle l',Z'\rangle \Longrightarrow Z \subseteq W \Longrightarrow W \subseteq V \Longrightarrow \exists\ W'.\ A \vdash \langle l,\ W\rangle \rightsquigarrow_\alpha* \langle l',W'\rangle \wedge Z' \subseteq W'$
⟨*proof*⟩

lemma *steps-z-alpha-alt*:
$A \vdash \langle l,\ Z\rangle \rightsquigarrow_\alpha \langle l',\ Z'\rangle \Longrightarrow A \vdash \langle l',\ Z'\rangle \rightsquigarrow_\alpha* \langle l'',\ Z''\rangle \Longrightarrow A \vdash \langle l,\ Z\rangle \rightsquigarrow_\alpha* \langle l'',\ Z''\rangle$
⟨*proof*⟩

lemma *steps-z-alpha-complete*:
$A \vdash \langle l,\ Z\rangle \rightsquigarrow* \langle l',Z'\rangle \Longrightarrow$ *valid-abstraction* $A\ X\ k \Longrightarrow Z \subseteq V \Longrightarrow Z' \neq \{\}$
$\Longrightarrow \exists\ Z''.\ A \vdash \langle l,\ Z\rangle \rightsquigarrow_\alpha* \langle l',Z''\rangle \wedge Z' \subseteq Z''$
⟨*proof*⟩

lemma *steps-z-alpha-complete′*:
$A \vdash \langle l,\ Z\rangle \rightsquigarrow* \langle l',Z'\rangle \Longrightarrow$ *valid-abstraction* $A\ X\ k \Longrightarrow Z \subseteq V \Longrightarrow Z' \neq \{\}$
$\Longrightarrow \exists\ Z''.\ A \vdash \langle l,\ Z\rangle \rightsquigarrow_\alpha* \langle l',Z''\rangle \wedge Z'' \neq \{\}$
⟨*proof*⟩

end

end
theory *Approx-Beta*
imports *DBM-Zone-Semantics Regions-Beta Closure*
begin

8 Correctness of β-approximation from α-regions

Instantiating real

instantiation *real* :: *linordered-ab-monoid-add*
begin

definition
neutral-real: $\mathbf{1} = (0 :: real)$

instance ⟨*proof*⟩

end

Merging the locales for the two types of regions

locale *Regions* =
fixes X **and** $k :: 'c \Rightarrow nat$ **and** $v :: 'c \Rightarrow nat$ **and** $n :: nat$ **and** *not-in-X*
assumes *finite*: *finite X*
assumes *clock-numbering*: *clock-numbering′ v n* $\forall k \leq n.\ k > 0 \longrightarrow (\exists c \in X.\ v\ c = k)$
$\forall\ c \in X.\ v\ c \leq n$
assumes *not-in-X*: *not-in-X* $\notin X$
assumes *non-empty*: $X \neq \{\}$
begin

definition $\mathcal{R}$*-def*: $\mathcal{R} \equiv \{$*Regions.region X I r* | *I r. Regions.valid-region X k I r*$\}$
definition $\mathcal{R}_\beta$*-def*: $\mathcal{R}_\beta \equiv \{$*Regions-Beta.region X I J r* | *I J r. Regions-Beta.valid-region X k I J r*$\}$
definition *V-def*: $V \equiv \{v\ .\ \forall\ x \in X.\ v\ x \geq 0\}$

sublocale *alpha-interp*: *AlphaClosure X k* $\mathcal{R}$ *V* ⟨*proof*⟩

sublocale *beta-interp*: *Beta-Regions′ X k* $\mathcal{R}_\beta$ *V v n not-in-X*
⟨*proof*⟩

abbreviation $Approx_\beta \equiv$ *beta-interp.*$Approx_\beta$

8.1 Preparing Bouyer's Theorem

lemma *region-dbm*:
assumes $R \in \mathcal{R}$
defines $v' \equiv \lambda\ i.$ *THE c.* $c \in X \wedge v\ c = i$
obtains M
where $[M]_{v,n} = R$

and $\forall\ i \leq n.\ \forall\ j \leq n.\ M\ i\ 0 = \infty \wedge j > 0 \wedge i \neq j \longrightarrow M\ i\ j = \infty \wedge M\ j\ i = \infty$
and $\forall\ i \leq n.\ M\ i\ i = Le\ 0$
and $\forall\ i \leq n.\ \forall\ j \leq n.\ i > 0 \wedge j > 0 \wedge M\ i\ 0 \neq \infty \wedge M\ j\ 0 \neq \infty \longrightarrow (\exists\ d :: int.$
$(-\ k\ (v'\ j) \leq d \wedge d \leq k\ (v'\ i) \wedge M\ i\ j = Le\ d \wedge M\ j\ i = Le\ (-d))$
$\vee\ (-\ k\ (v'\ j) \leq d - 1 \wedge d \leq k\ (v'\ i) \wedge M\ i\ j = Lt\ d \wedge M\ j\ i = Lt\ (-d + 1)))$
and $\forall\ i \leq n.\ i > 0 \wedge M\ i\ 0 \neq \infty \longrightarrow$
$(\exists\ d :: int.\ d \leq k\ (v'\ i) \wedge d \geq 0$
$\wedge\ (M\ i\ 0 = Le\ d \wedge M\ 0\ i = Le\ (-d) \vee M\ i\ 0 = Lt\ d \wedge M\ 0\ i = Lt\ (-d + 1)))$
and $\forall\ i \leq n.\ i > 0 \longrightarrow (\exists\ d :: int.\ -\ k\ (v'\ i) \leq d \wedge d \leq 0 \wedge (M\ 0\ i = Le\ d \vee M\ 0\ i = Lt\ d))$
and $\forall\ i.\ \forall\ j.\ M\ i\ j \neq \infty \longrightarrow get\text{-}const\ (M\ i\ j) \in \mathbb{Z}$
and $\forall\ i \leq n.\ \forall\ j \leq n.\ M\ i\ j \neq \infty \wedge i > 0 \wedge j > 0 \longrightarrow$
$(\exists\ d::\ int.\ (M\ i\ j = Le\ d \vee M\ i\ j = Lt\ d) \wedge (-\ k\ (v'\ j)) \leq d \wedge d \leq k\ (v'\ i))$
⟨*proof*⟩

lemma *len-inf-elem*:
$(a,\ b) \in set\ (arcs\ i\ j\ xs) \Longrightarrow M\ a\ b = \infty \Longrightarrow len\ M\ i\ j\ xs = \infty$
⟨*proof*⟩

lemma *dbm-add-strict-right-mono-neutral*: $a < Le\ d \Longrightarrow a + Le\ (-d) < Le\ 0$
⟨*proof*⟩

lemma *dbm-lt-not-inf-less*[*intro*]: $A \neq \infty \Longrightarrow A \prec \infty$ ⟨*proof*⟩

lemma *add-inf*[*simp*]:
$a + \infty = \infty\ \infty + a = \infty$
⟨*proof*⟩

lemma *inf-lt*[*simp*,*dest*!]:
$\infty < x \Longrightarrow False$
⟨*proof*⟩

lemma *zone-diag-lt*:
assumes $a \leq n\ b \leq n$ **and** C: $v\ c1 = a\ v\ c2 = b$ **and** *not0*: $a > 0\ b > 0$
shows $[(\lambda\ i\ j.\ if\ i = a \wedge j = b\ then\ Lt\ d\ else\ \infty)]_{v,n} = \{u.\ u\ c1 - u\ c2 < d\}$
⟨*proof*⟩

lemma *zone-diag-le*:
assumes $a \leq n$ $b \leq n$ **and** *C*: $v\ c1 = a\ v\ c2 = b$ **and** *not0*: $a > 0\ b > 0$
shows $[(\lambda\ i\ j.\ \mathit{if}\ i = a \wedge j = b\ \mathit{then}\ \mathit{Le}\ d\ \mathit{else}\ \infty)]_{v,n} = \{u.\ u\ c1 - u\ c2 \leq d\}$
⟨*proof*⟩

lemma *zone-diag-lt-2*:
assumes $a \leq n$ **and** *C*: $v\ c = a$ **and** *not0*: $a > 0$
shows $[(\lambda\ i\ j.\ \mathit{if}\ i = a \wedge j = 0\ \mathit{then}\ \mathit{Lt}\ d\ \mathit{else}\ \infty)]_{v,n} = \{u.\ u\ c < d\}$
⟨*proof*⟩

lemma *zone-diag-le-2*:
assumes $a \leq n$ **and** *C*: $v\ c = a$ **and** *not0*: $a > 0$
shows $[(\lambda\ i\ j.\ \mathit{if}\ i = a \wedge j = 0\ \mathit{then}\ \mathit{Le}\ d\ \mathit{else}\ \infty)]_{v,n} = \{u.\ u\ c \leq d\}$
⟨*proof*⟩

lemma *zone-diag-lt-3*:
assumes $a \leq n$ **and** *C*: $v\ c = a$ **and** *not0*: $a > 0$
shows $[(\lambda\ i\ j.\ \mathit{if}\ i = 0 \wedge j = a\ \mathit{then}\ \mathit{Lt}\ d\ \mathit{else}\ \infty)]_{v,n} = \{u.\ -\ u\ c < d\}$
⟨*proof*⟩

lemma *len-int-closed*:
$\forall\ i\ j.\ (M\ i\ j :: \mathit{real}) \in \mathbb{Z} \Longrightarrow \mathit{len}\ M\ i\ j\ xs \in \mathbb{Z}$
⟨*proof*⟩

lemma *get-const-distr*:
$a \neq \infty \Longrightarrow b \neq \infty \Longrightarrow \mathit{get\text{-}const}\ (a + b) = \mathit{get\text{-}const}\ a + \mathit{get\text{-}const}\ b$
⟨*proof*⟩

lemma *len-int-dbm-closed*:
$\forall\ (i, j) \in \mathit{set}\ (\mathit{arcs}\ i\ j\ xs).\ (\mathit{get\text{-}const}\ (M\ i\ j) :: \mathit{real}) \in \mathbb{Z} \wedge M\ i\ j \neq \infty$
$\Longrightarrow \mathit{get\text{-}const}\ (\mathit{len}\ M\ i\ j\ xs) \in \mathbb{Z} \wedge \mathit{len}\ M\ i\ j\ xs \neq \infty$
⟨*proof*⟩

lemma *zone-diag-le-3*:
assumes $a \leq n$ **and** *C*: $v\ c = a$ **and** *not0*: $a > 0$
shows $[(\lambda\ i\ j.\ \mathit{if}\ i = 0 \wedge j = a\ \mathit{then}\ \mathit{Le}\ d\ \mathit{else}\ \infty)]_{v,n} = \{u.\ -\ u\ c \leq d\}$
⟨*proof*⟩

lemma *dbm-lt'*:
assumes $[M]_{v,n} \subseteq V\ M\ a\ b \leq \mathit{Lt}\ d\ a \leq n\ b \leq n\ v\ c1 = a\ v\ c2 = b\ a > 0\ b > 0$
shows $[M]_{v,n} \subseteq \{u \in V.\ u\ c1 - u\ c2 < d\}$
⟨*proof*⟩

lemma *dbm-lt'2*:
assumes $[M]_{v,n} \subseteq V\ M\ a\ 0 \leq Lt\ d\ a \leq n\ v\ c1 = a\ a > 0$
shows $[M]_{v,n} \subseteq \{u \in V.\ u\ c1 < d\}$
⟨*proof*⟩

lemma *dbm-lt'3*:
assumes $[M]_{v,n} \subseteq V\ M\ 0\ a \leq Lt\ d\ a \leq n\ v\ c1 = a\ a > 0$
shows $[M]_{v,n} \subseteq \{u \in V.\ -\ u\ c1 < d\}$
⟨*proof*⟩

lemma *dbm-le'*:
assumes $[M]_{v,n} \subseteq V\ M\ a\ b \leq Le\ d\ a \leq n\ b \leq n\ v\ c1 = a\ v\ c2 = b\ a > 0\ b > 0$
shows $[M]_{v,n} \subseteq \{u \in V.\ u\ c1\ -\ u\ c2 \leq d\}$
⟨*proof*⟩

lemma *dbm-le'2*:
assumes $[M]_{v,n} \subseteq V\ M\ a\ 0 \leq Le\ d\ a \leq n\ v\ c1 = a\ a > 0$
shows $[M]_{v,n} \subseteq \{u \in V.\ u\ c1 \leq d\}$
⟨*proof*⟩

lemma *dbm-le'3*:
assumes $[M]_{v,n} \subseteq V\ M\ 0\ a \leq Le\ d\ a \leq n\ v\ c1 = a\ a > 0$
shows $[M]_{v,n} \subseteq \{u \in V.\ -\ u\ c1 \leq d\}$
⟨*proof*⟩

lemma *int-zone-dbm*:
assumes $\forall\ (\text{-},d) \in$ *collect-clock-pairs cc*. $d \in \mathbb{Z} \forall\ c \in$ *collect-clks cc*. $v\ c \leq n$
obtains M **where** $\{u.\ u \vdash cc\} = [M]_{v,n}$ **and** *dbm-int M n*
⟨*proof*⟩

lemma *non-empty-dbm-diag-set'*:
assumes *clock-numbering' v n* $\forall\, i{\leq}n.\ \forall\, j{\leq}n.\ M\ i\ j \neq \infty \longrightarrow$ *get-const* $(M\ i\ j) \in \mathbb{Z}$
$[M]_{v,n} \neq \{\}$
obtains M' **where** $[M]_{v,n} = [M']_{v,n} \wedge (\forall\, i{\leq}n.\ \forall\, j{\leq}n.\ M'\ i\ j \neq \infty \longrightarrow$ *get-const* $(M'\ i\ j) \in \mathbb{Z})$
$\wedge\ (\forall\ i \leq n.\ M'\ i\ i = \mathbf{1})$
⟨*proof*⟩

lemma *dbm-entry-int*:
$x \neq \infty \Longrightarrow$ *get-const* $x \in \mathbb{Z} \Longrightarrow \exists\ d ::$ *int*. $x = Le\ d \vee x = Lt\ d$

⟨*proof*⟩

abbreviation *vabstr* ≡ *beta-interp.vabstr*

8.2 Bouyer's Main Theorem

theorem *region-zone-intersect-empty-approx-correct*:
  assumes $R \in \mathcal{R}$ $Z \subseteq V$ $R \cap Z = \{\}$ *vabstr Z M*
  shows $R \cap Approx_\beta\ Z = \{\}$
⟨*proof*⟩

8.3 Nice Corollaries of Bouyer's Theorem

lemma $\mathcal{R}$-V: $\bigcup\ \mathcal{R} = V$ ⟨*proof*⟩

lemma *regions-beta-V*: $R \in \mathcal{R}_\beta \Longrightarrow R \subseteq V$ ⟨*proof*⟩

lemma *apx-V*: $Z \subseteq V \Longrightarrow Approx_\beta\ Z \subseteq V$
⟨*proof*⟩

corollary *approx-β-closure-α*:
  assumes $Z \subseteq V$ *vabstr Z M*
  shows $Approx_\beta\ Z \subseteq Closure_\alpha\ Z$
⟨*proof*⟩

definition $V' \equiv \{Z.\ Z \subseteq V \wedge (\exists\ M.\ vabstr\ Z\ M)\}$

corollary *approx-β-closure-α'*: $Z \in V' \Longrightarrow Approx_\beta\ Z \subseteq Closure_\alpha\ Z$
⟨*proof*⟩

We could prove this more directly too (without using $Closure_\alpha\ Z$), obviously

lemma *apx-empty-iff*:
  assumes $Z \subseteq V$ *vabstr Z M*
  shows $Z = \{\} \longleftrightarrow Approx_\beta\ Z = \{\}$
⟨*proof*⟩

lemma *apx-empty-iff'*:
  assumes $Z \in V'$ **shows** $Z = \{\} \longleftrightarrow Approx_\beta\ Z = \{\}$
⟨*proof*⟩

lemma *apx-V'*:
  assumes $Z \subseteq V$ **shows** $Approx_\beta\ Z \in V'$
⟨*proof*⟩

8.4 A New Zone Semantics Abstracting with $Approx_\beta$

lemma *step-z-V'*:
assumes $A \vdash \langle l,Z\rangle \rightsquigarrow \langle l',Z'\rangle$ *valid-abstraction* $A\ X\ k\ \forall\, c{\in}clk\text{-}set\ A.\ v\ c \leq n\ Z \in V'$
shows $Z' \in V'$
⟨*proof*⟩

lemma *steps-z-V'*:
$A \vdash \langle l,Z\rangle \rightsquigarrow* \langle l',Z'\rangle \Longrightarrow$ *valid-abstraction* $A\ X\ k \Longrightarrow \forall\, c{\in}clk\text{-}set\ A.\ v\ c \leq n \Longrightarrow Z \in V' \Longrightarrow Z' \in V'$
⟨*proof*⟩

8.4.1 Single Step

inductive *step-z-beta* ::
$('a, 'c, t, 's)\ ta \Rightarrow 's \Rightarrow ('c, t)\ zone \Rightarrow 's \Rightarrow ('c, t)\ zone \Rightarrow bool$
(- ⊢ ⟨-, -⟩ $\rightsquigarrow_\beta$ ⟨-, -⟩ [61,61,61] 61)
where
step-beta: $A \vdash \langle l, Z\rangle \rightsquigarrow \langle l', Z'\rangle \Longrightarrow A \vdash \langle l, Z\rangle \rightsquigarrow_\beta \langle l', Approx_\beta\ Z'\rangle$

inductive-cases[*elim*!]: $A \vdash \langle l, u\rangle \rightsquigarrow_\beta \langle l',u'\rangle$

declare *step-z-beta.intros*[*intro*]

lemma *step-z-alpha-sound*:
$A \vdash \langle l, Z\rangle \rightsquigarrow_\beta \langle l',Z'\rangle \Longrightarrow$ *valid-abstraction* $A\ X\ k \Longrightarrow \forall\, c{\in}clk\text{-}set\ A.\ v\ c \leq n \Longrightarrow Z \in V' \Longrightarrow Z' \neq \{\}$
$\Longrightarrow \exists\ Z''.\ A \vdash \langle l, Z\rangle \rightsquigarrow \langle l',Z''\rangle \wedge Z'' \neq \{\}$
⟨*proof*⟩

lemma *step-z-alpha-complete*:
$A \vdash \langle l, Z\rangle \rightsquigarrow \langle l',Z'\rangle \Longrightarrow$ *valid-abstraction* $A\ X\ k \Longrightarrow \forall\, c{\in}clk\text{-}set\ A.\ v\ c \leq n \Longrightarrow Z \in V' \Longrightarrow Z' \neq \{\}$
$\Longrightarrow \exists\ Z''.\ A \vdash \langle l, Z\rangle \rightsquigarrow_\beta \langle l', Z''\rangle \wedge Z'' \neq \{\}$
⟨*proof*⟩

8.4.2 Multi step

inductive
steps-z-beta :: $('a, 'c, t, 's)\ ta \Rightarrow 's \Rightarrow ('c, t)\ zone \Rightarrow 's \Rightarrow ('c, t)\ zone \Rightarrow bool$
(- ⊢ ⟨-, -⟩ $\rightsquigarrow_\beta$* ⟨-, -⟩ [61,61,61] 61)
where
refl: $A \vdash \langle l, Z\rangle \rightsquigarrow_\beta* \langle l, Z\rangle$ |

step: $A \vdash \langle l,\ Z\rangle \rightsquigarrow_\beta* \langle l',\ Z'\rangle \Longrightarrow A \vdash \langle l',\ Z'\rangle \rightsquigarrow_\beta \langle l'',\ Z''\rangle \Longrightarrow A \vdash \langle l,\ Z\rangle \rightsquigarrow_\beta* \langle l'',\ Z''\rangle$

declare *steps-z-beta.intros*[*intro*]

lemma *V'-V*: $Z \in V' \Longrightarrow Z \subseteq V$ ⟨*proof*⟩

lemma *steps-z-beta-V'*:
$A \vdash \langle l,\ Z\rangle \rightsquigarrow_\beta* \langle l',\ Z'\rangle \Longrightarrow$ *valid-abstraction* $A\ X\ k \Longrightarrow \forall\, c{\in}$*clk-set* $A.\ v\ c \leq n \Longrightarrow Z \in V' \Longrightarrow Z' \in V'$
⟨*proof*⟩

lemma *alpha-beta-step*:
$A \vdash \langle l,\ Z\rangle \rightsquigarrow_\beta \langle l',\ Z'\rangle \Longrightarrow$ *valid-abstraction* $A\ X\ k \Longrightarrow \forall\, c{\in}$*clk-set* $A.\ v\ c \leq n \Longrightarrow Z \in V'$
$\Longrightarrow \exists\ Z''.\ A \vdash \langle l,\ Z\rangle \rightsquigarrow_\alpha \langle l',\ Z''\rangle \wedge Z' \subseteq Z''$
⟨*proof*⟩

Soundness **lemma** *alpha-beta-step'*:
$A \vdash \langle l,\ Z\rangle \rightsquigarrow_\beta \langle l',\ Z'\rangle \Longrightarrow$ *valid-abstraction* $A\ X\ k \Longrightarrow \forall\, c{\in}$*clk-set* $A.\ v\ c \leq n \Longrightarrow Z \in V' \Longrightarrow W \subseteq V$
$\Longrightarrow Z \subseteq W \Longrightarrow \exists\ W'.\ A \vdash \langle l,\ W\rangle \rightsquigarrow_\alpha \langle l',\ W'\rangle \wedge Z' \subseteq W'$
⟨*proof*⟩

lemma *alpha-beta-steps*:
$A \vdash \langle l,\ Z\rangle \rightsquigarrow_\beta* \langle l',\ Z'\rangle \Longrightarrow$ *valid-abstraction* $A\ X\ k \Longrightarrow \forall\, c{\in}$*clk-set* $A.\ v\ c \leq n \Longrightarrow Z \in V'$
$\Longrightarrow \exists\ Z''.\ A \vdash \langle l,\ Z\rangle \rightsquigarrow_\alpha* \langle l',\ Z''\rangle \wedge Z' \subseteq Z''$
⟨*proof*⟩

corollary *steps-z-beta-sound*:
$A \vdash \langle l,\ Z\rangle \rightsquigarrow_\beta* \langle l',\ Z'\rangle \Longrightarrow \forall\, c{\in}$*clk-set* $A.\ v\ c \leq n \Longrightarrow$ *valid-abstraction* $A\ X\ k \Longrightarrow Z \in V' \Longrightarrow Z' \neq \{\}$
$\Longrightarrow \exists\ Z''.\ A \vdash \langle l,\ Z\rangle \rightsquigarrow* \langle l',\ Z''\rangle \wedge Z'' \neq \{\}$
⟨*proof*⟩

Completeness **lemma** *apx-mono*:
$Z' \subseteq V \Longrightarrow Z \subseteq Z' \Longrightarrow Approx_\beta\ Z \subseteq Approx_\beta\ Z'$
⟨*proof*⟩

lemma *step-z-beta-mono*:
$A \vdash \langle l,\ Z\rangle \rightsquigarrow_\beta \langle l',\ Z'\rangle \Longrightarrow Z \subseteq W \Longrightarrow W \subseteq V \Longrightarrow \exists\ W'.\ A \vdash \langle l,\ W\rangle \rightsquigarrow_\beta \langle l',\ W'\rangle \wedge Z' \subseteq W'$

⟨*proof*⟩

lemma *steps-z-beta-V*: $A \vdash \langle l, Z\rangle \rightsquigarrow_{\beta *} \langle l', Z'\rangle \Longrightarrow Z \subseteq V \Longrightarrow Z' \subseteq V$
⟨*proof*⟩

lemma *steps-z-beta-mono*:
$A \vdash \langle l, Z\rangle \rightsquigarrow_{\beta *} \langle l', Z'\rangle \Longrightarrow Z \subseteq W \Longrightarrow W \subseteq V \Longrightarrow \exists\ W'.\ A \vdash \langle l, W\rangle \rightsquigarrow_{\beta *} \langle l', W'\rangle \wedge Z' \subseteq W'$
⟨*proof*⟩

lemma *steps-z-beta-alt*:
$A \vdash \langle l, Z\rangle \rightsquigarrow_{\beta} \langle l', Z'\rangle \Longrightarrow A \vdash \langle l', Z'\rangle \rightsquigarrow_{\beta *} \langle l'', Z''\rangle \Longrightarrow A \vdash \langle l, Z\rangle \rightsquigarrow_{\beta *} \langle l'', Z''\rangle$
⟨*proof*⟩

lemma *steps-z-beta-complete*:
$A \vdash \langle l, Z\rangle \rightsquigarrow * \langle l', Z'\rangle \Longrightarrow$ *valid-abstraction* $A\ X\ k \Longrightarrow Z \subseteq V$
$\Longrightarrow \exists\ Z''.\ A \vdash \langle l, Z\rangle \rightsquigarrow_{\beta *} \langle l', Z''\rangle \wedge Z' \subseteq Z''$
⟨*proof*⟩

lemma *steps-z-beta-complete'*:
$A \vdash \langle l, Z\rangle \rightsquigarrow * \langle l', Z'\rangle \Longrightarrow$ *valid-abstraction* $A\ X\ k \Longrightarrow Z \subseteq V \Longrightarrow Z' \neq \{\}$
$\Longrightarrow \exists\ Z''.\ A \vdash \langle l, Z\rangle \rightsquigarrow_{\beta *} \langle l', Z''\rangle \wedge Z'' \neq \{\}$
⟨*proof*⟩

end

end

9 Forward Analysis with DBMs and Widening

theory *Normalized-Zone-Semantics*
imports *DBM-Zone-Semantics Approx-Beta*
begin

9.1 DBM-based Semantics with Normalization

9.1.1 Single Step

inductive *step-z-norm* ::
$('a, 'c, t, 's)\ ta \Rightarrow 's \Rightarrow t\ DBM \Rightarrow (nat \Rightarrow nat) \Rightarrow ('c \Rightarrow nat) \Rightarrow nat \Rightarrow 's \Rightarrow t\ DBM \Rightarrow bool$
$(\text{-} \vdash \langle \text{-}, \text{-}\rangle \rightsquigarrow_{\text{-},\text{-},\text{-}} \langle \text{-}, \text{-}\rangle\ [61,61,61,61,61]\ 61)$

where *step-z-norm*:
$A \vdash \langle l,D\rangle \rightsquigarrow_{v,n} \langle l',\ D'\rangle \Longrightarrow A \vdash \langle l,D\rangle \rightsquigarrow_{k,v,n} \langle l',\ norm\ (FW\ D'\ n)\ k\ n\rangle$

inductive *steps-z-norm* ::
$('a,\ 'c,\ t,\ 's)\ ta \Rightarrow 's \Rightarrow t\ DBM \Rightarrow (nat \Rightarrow nat) \Rightarrow ('c \Rightarrow nat) \Rightarrow nat \Rightarrow$
$'s \Rightarrow t\ DBM \Rightarrow bool$
$(\text{-} \vdash \langle \text{-},\ \text{-}\rangle \rightsquigarrow_{\text{-},\text{-},\text{-}}* \langle \text{-},\ \text{-}\rangle\ [61,61,61,61,61]\ 61)$
where
refl: $A \vdash \langle l,\ Z\rangle \rightsquigarrow_{k,v,n}* \langle l,\ Z\rangle$ |
step: $A \vdash \langle l,\ Z\rangle \rightsquigarrow_{k,v,n}* \langle l',\ Z'\rangle \Longrightarrow A \vdash \langle l',\ Z'\rangle \rightsquigarrow_{k,v,n} \langle l'',\ Z''\rangle$
$\Longrightarrow A \vdash \langle l,\ Z\rangle \rightsquigarrow_{k,v,n}* \langle l'',\ Z''\rangle$

context *Regions*
begin

abbreviation $v' \equiv$ *beta-interp.v'*

abbreviation *step-z-norm'* $(\text{-} \vdash \langle \text{-},\ \text{-}\rangle \rightsquigarrow_{\mathcal{N}} \langle \text{-},\ \text{-}\rangle\ [61,61,61]\ 61)$
where
$A \vdash \langle l,\ D\rangle \rightsquigarrow_{\mathcal{N}} \langle l',\ D'\rangle \equiv A \vdash \langle l,\ D\rangle \rightsquigarrow_{(k\ o\ v'),v,n} \langle l',\ D'\rangle$

abbreviation *steps-z-norm'* $(\text{-} \vdash \langle \text{-},\ \text{-}\rangle \rightsquigarrow_{\mathcal{N}}* \langle \text{-},\ \text{-}\rangle\ [61,61,61]\ 61)$
where
$A \vdash \langle l,\ D\rangle \rightsquigarrow_{\mathcal{N}}* \langle l',\ D'\rangle \equiv A \vdash \langle l,\ D\rangle \rightsquigarrow_{(k\ o\ v'),v,n}* \langle l',\ D'\rangle$

inductive-cases[*elim*!]: $A \vdash \langle l,\ u\rangle \rightsquigarrow_{\mathcal{N}} \langle l',u'\rangle$

declare *step-z-norm.intros*[*intro*]

lemma *apx-empty-iff''*:
assumes *canonical M1 n* $[M1]_{v,n} \subseteq V$ *dbm-int M1 n*
shows $[M1]_{v,n} = \{\} \longleftrightarrow [norm\ M1\ (k\ o\ v')\ n]_{v,n} = \{\}$
⟨*proof*⟩

inductive *valid-dbm* **where**
$[M]_{v,n} \subseteq V \Longrightarrow dbm\text{-}int\ M\ n \Longrightarrow valid\text{-}dbm\ M$

inductive-cases *valid-dbm-cases*[*elim*]: *valid-dbm M*

declare *valid-dbm.intros*[*intro*]

lemma *step-z-valid-dbm*:
assumes $A \vdash \langle l,\ D\rangle \rightsquigarrow_{v,n} \langle l',\ D'\rangle$
and *global-clock-numbering A v n valid-abstraction A X k valid-dbm D*

shows *valid-dbm* D'
⟨*proof*⟩

lemma *FW-zone-equiv-spec*:
shows $[M]_{v,n} = [FW\ M\ n]_{v,n}$
⟨*proof*⟩

lemma *cn-weak*: $\forall\, k{\leq}n.\ 0 < k \longrightarrow (\exists\, c.\ v\ c = k)$ ⟨*proof*⟩

lemma *valid-dbm-non-empty-diag*:
assumes *valid-dbm* M $[M]_{v,n} \neq \{\}$
shows $\forall\ k \leq n.\ M\ k\ k \geq \mathbf{1}$
⟨*proof*⟩

lemma *non-empty-cycle-free*:
assumes $[M]_{v,n} \neq \{\}$
shows *cycle-free* $M\ n$
⟨*proof*⟩

lemma *norm-empty-diag-preservation*:
assumes $i \leq n$
assumes $M\ i\ i < Le\ 0$
shows *norm* $M\ (k\ o\ v')\ n\ i\ i < Le\ 0$
⟨*proof*⟩

lemma *norm-FW-empty*:
assumes *valid-dbm* M
assumes $[M]_{v,n} = \{\}$
shows $[norm\ (FW\ M\ n)\ (k\ o\ v')\ n]_{v,n} = \{\}$ (**is** $[?M]_{v,n} = \{\}$)
⟨*proof*⟩

lemma *apx-norm-eq-spec*:
assumes *valid-dbm* M
and $[M]_{v,n} \neq \{\}$
shows *beta-interp.Approx*$_\beta$ $([M]_{v,n}) = [norm\ (FW\ M\ n)\ (k\ o\ v')\ n]_{v,n}$
⟨*proof*⟩

print-statement *step-z-norm.inducts*

lemma *step-z-norm-induct*[*case-names - step-z-norm step-z-refl*]:
assumes *x1* $\vdash \langle x2,\ x3\rangle \rightsquigarrow_{(k\ o\ v'),v,n} \langle x7,x8\rangle$
and *step-z-norm*:
$\bigwedge A\ l\ D\ l'\ D'.$

$A \vdash \langle l,\ D\rangle \rightsquigarrow_{v,n} \langle l',D'\rangle \Longrightarrow$
$P\ A\ l\ D\ l'\ (norm\ (FW\ D'\ n)\ (k\ o\ v')\ n)$
shows *P x1 x2 x3 x7 x8*
⟨*proof*⟩

lemma *FW-valid-preservation*:
assumes *valid-dbm M*
shows *valid-dbm* $(FW\ M\ n)$
⟨*proof*⟩

Obsolete

lemma *norm-diag-preservation*:
assumes $\forall\, l{\le}n.\ M1\ l\ l \le \mathbf{1}$
shows $\forall\, l{\le}n.\ (norm\ M1\ (k\ o\ v')\ n)\ l\ l \le \mathbf{1}$ (**is** $\forall\ \ l \le n.\ ?M\ l\ l \le \mathbf{1}$)
⟨*proof*⟩

lemma *norm-FW-valid-preservation-non-empty*:
assumes *valid-dbm M* $[M]_{v,n} \neq \{\}$
shows *valid-dbm* $(norm\ (FW\ M\ n)\ (k\ o\ v')\ n)$ (**is** *valid-dbm ?M*)
⟨*proof*⟩

lemma *norm-FW-valid-preservation-empty*:
assumes *valid-dbm M* $[M]_{v,n} = \{\}$
shows *valid-dbm* $(norm\ (FW\ M\ n)\ (k\ o\ v')\ n)$ (**is** *valid-dbm ?M*)
⟨*proof*⟩

lemma *norm-FW-valid-preservation*:
assumes *valid-dbm M*
shows *valid-dbm* $(norm\ (FW\ M\ n)\ (k\ o\ v')\ n)$
⟨*proof*⟩

lemma *norm-beta-sound*:
assumes $A \vdash \langle l,D\rangle \rightsquigarrow_{\mathcal{N}} \langle l',D'\rangle$ *global-clock-numbering A v n valid-abstraction A X k*
and *valid-dbm D*
shows $A \vdash \langle l,[D]_{v,n}\rangle \rightsquigarrow_{\beta} \langle l',[D']_{v,n}\rangle$ ⟨*proof*⟩

lemma *step-z-norm-valid-dbm*:
assumes $A \vdash \langle l,\ D\rangle \rightsquigarrow_{\mathcal{N}} \langle l',D'\rangle$ *global-clock-numbering A v n valid-abstraction A X k valid-dbm D*
shows *valid-dbm D'* ⟨*proof*⟩

lemma *norm-beta-complete*:
assumes $A \vdash \langle l,[D]_{v,n}\rangle \rightsquigarrow_{\beta} \langle l',Z\rangle$ *global-clock-numbering A v n valid-abstraction*

$A\ X\ k$
and *valid-dbm D*
obtains D' **where** $A \vdash \langle l,D\rangle \rightsquigarrow_{\mathcal{N}} \langle l',D'\rangle\ [D']_{v,n} = Z$ *valid-dbm* D'
⟨*proof*⟩

9.1.2 Multi step

declare *steps-z-norm.intros*[*intro*]

lemma *steps-z-norm-induct*[*case-names - refl step*]:
assumes $x1 \vdash \langle x2,\ x3\rangle \rightsquigarrow_{(k\ o\ v'),v,n}{}^{*} \langle x7,x8\rangle$
and $\bigwedge A\ l\ Z.\ P\ A\ l\ Z\ l\ Z$
and
$\bigwedge A\ l\ Z\ l'\ Z'\ l''\ Z''.$
$A \vdash \langle l,\ Z\rangle \rightsquigarrow_{(k\ o\ v'),v,n}{}^{*} \langle l',Z'\rangle \Longrightarrow$
$P\ A\ l\ Z\ l'\ Z' \Longrightarrow$
$A \vdash \langle l',\ Z'\rangle \rightsquigarrow_{(k\ o\ v'),v,n} \langle l'',Z''\rangle \Longrightarrow P\ A\ l\ Z\ l''\ Z''$
shows $P\ x1\ x2\ x3\ x7\ x8$
⟨*proof*⟩

lemma *norm-beta-sound-multi*:
assumes $A \vdash \langle l,D\rangle \rightsquigarrow_{\mathcal{N}}{}^{*} \langle l',D'\rangle$ *global-clock-numbering A v n valid-abstraction A X k valid-dbm D*
shows $A \vdash \langle l,[D]_{v,n}\rangle \rightsquigarrow_{\beta}{}^{*} \langle l',[D']_{v,n}\rangle \wedge$ *valid-dbm* D' ⟨*proof*⟩

lemma *norm-beta-complete-multi*:
assumes $A \vdash \langle l,[D]_{v,n}\rangle \rightsquigarrow_{\beta}{}^{*} \langle l',Z\rangle$ *global-clock-numbering A v n valid-abstraction A X k*
and *valid-dbm D*
obtains D' **where** $A \vdash \langle l,D\rangle \rightsquigarrow_{\mathcal{N}}{}^{*} \langle l',D'\rangle\ [D']_{v,n} = Z$ *valid-dbm* D'
⟨*proof*⟩

lemma *norm-beta-equiv-multi*:
assumes *global-clock-numbering A v n valid-abstraction A X k*
and *valid-dbm D*
shows $(\exists\ D'.\ A \vdash \langle l,D\rangle \rightsquigarrow_{\mathcal{N}}{}^{*} \langle l',D'\rangle \wedge Z = [D']_{v,n}) \longleftrightarrow A \vdash \langle l,[D]_{v,n}\rangle \rightsquigarrow_{\beta}{}^{*} \langle l',Z\rangle$
⟨*proof*⟩

9.1.3 Connecting with Correctness Results for Approximating Semantics

lemma *steps-z-norm-complete'*:
assumes $A \vdash \langle l,[D]_{v,n}\rangle \rightsquigarrow^{*} \langle l',Z\rangle$ *global-clock-numbering A v n valid-abstraction*

A X k
and *valid-dbm D*
shows $\exists\ D'.\ A \vdash \langle l,\ D\rangle \rightsquigarrow_{\mathcal{N}} * \langle l',D'\rangle \wedge\ \ Z \subseteq [D']_{v,n}$
⟨*proof*⟩

lemma *valid-dbm-V'*:
assumes *valid-dbm M*
shows $[M]_{v,n} \in V'$
⟨*proof*⟩

lemma *steps-z-norm-sound'*:
assumes $A \vdash \langle l,D\rangle \rightsquigarrow_{\mathcal{N}} * \langle l',D'\rangle$
and *global-clock-numbering A v n*
and *valid-abstraction A X k*
and *valid-dbm D*
and $[D']_{v,n} \neq \{\}$
shows $\exists Z.\ A \vdash \langle l,[D]_{v,n}\rangle \rightsquigarrow * \langle l',Z\rangle \wedge Z \neq \{\}$
⟨*proof*⟩

9.2 The Final Result About Language Emptiness

lemma *steps-z-norm-complete*:
assumes $A \vdash \langle l,\ u\rangle \rightarrow * \langle l',\ u'\rangle\ u \in [D]_{v,n}$
and *global-clock-numbering A v n valid-abstraction A X k valid-dbm D*
shows $\exists\ D'.\ A \vdash \langle l,\ D\rangle \rightsquigarrow_{\mathcal{N}} * \langle l',D'\rangle \wedge u' \in [D']_{v,n}$
⟨*proof*⟩

lemma *steps-z-norm-sound*:
assumes $A \vdash \langle l,D\rangle \rightsquigarrow_{\mathcal{N}} * \langle l',D'\rangle$
and *global-clock-numbering A v n valid-abstraction A X k valid-dbm D*
and $[D']_{v,n} \neq \{\}$
shows $\exists\ u \in [D]_{v,n}.\ \exists\ u'.\ A \vdash \langle l,\ u\rangle \rightarrow * \langle l',\ u'\rangle$
⟨*proof*⟩

theorem *steps-z-norm-decides-emptiness*:
assumes *global-clock-numbering A v n valid-abstraction A X k valid-dbm D*
shows $(\exists\ D'.\ A \vdash \langle l,\ D\rangle \rightsquigarrow_{\mathcal{N}} * \langle l',D'\rangle \wedge [D']_{v,n} \neq \{\})$
$\longleftrightarrow (\exists\ u \in [D]_{v,n}.\ \exists\ u'.\ A \vdash \langle l,\ u\rangle \rightarrow * \langle l',\ u'\rangle)$
⟨*proof*⟩

9.3 Finiteness of the Search Space

abbreviation *dbm-default M* ≡ (∀ *i* > *n*. ∀ *j*. *M i j* = **1**) ∧ (∀ *j* > *n*. ∀ *i*. *M i j* = **1**)

lemma *a* ∈ ℤ ⟹ ∃ *b*. *a* = *real-of-int b* ⟨*proof*⟩

lemma *norm-default-preservation*:
dbm-default M ⟹ *dbm-default* (*norm M* (*k o v'*) *n*)
⟨*proof*⟩

lemma *normalized-integral-dbms-finite*:
finite {*norm M* (*k o v'*) *n* | *M*. *dbm-int M n* ∧ *dbm-default M*}
⟨*proof*⟩

end

9.4 Appendix: Standard Clock Numberings for Concrete Models

locale *Regions'* =
fixes *X* **and** *k* :: *'c* ⇒ *nat* **and** *v* :: *'c* ⇒ *nat* **and** *n* :: *nat* **and** *not-in-X*
assumes *finite*: *finite X*
assumes *clock-numbering'*: ∀ *c* ∈ *X*. *v c* > *0* ∀ *c*. *c* ∉ *X* ⟶ *v c* > *n*
assumes *bij*: *bij-betw v X* {*1*..*n*}
assumes *non-empty*: *X* ≠ {}
assumes *not-in-X*: *not-in-X* ∉ *X*

begin

lemma *inj*: *inj-on v X* ⟨*proof*⟩

lemma *cn-weak*: ∀ *c*. *v c* > *0* ⟨*proof*⟩

lemma *in-X*: **assumes** *v x* ≤ *n* **shows** *x* ∈ *X* ⟨*proof*⟩

end

sublocale *Regions'* ⊆ *Regions*
⟨*proof*⟩

lemma *standard-abstraction*:
assumes *finite* (*clkp-set A*) *finite* (*collect-clkvt* (*trans-of A*)) ∀(-,*m*::*real*)

$\in$ *clkp-set A*. $m \in \mathbb{N}$
obtains $k :: 'c \Rightarrow nat$ **where** *valid-abstraction A* (*clk-set A*) k
⟨*proof*⟩

definition
finite-ta A ≡ *finite* (*clkp-set A*) ∧ *finite* (*collect-clkvt* (*trans-of A*))
∧ (∀ (-,*m*::*real*) ∈ *clkp-set A*. $m \in \mathbb{N}$) ∧ *clk-set A* ≠ {} ∧
−*clk-set A* ≠ {}

lemma *finite-ta-Regions'*:
fixes $A :: ('a, 'c, real, 's)$ *ta*
assumes *finite-ta A*
obtains *v n x* **where** *Regions'* (*clk-set A*) *v n x*
⟨*proof*⟩

lemma *finite-ta-RegionsD*:
assumes *finite-ta A*
obtains $k :: 'b \Rightarrow nat$ **and** *v n x* **where**
Regions' (*clk-set A*) *v n x valid-abstraction A* (*clk-set A*) *k global-clock-numbering*
A v n
⟨*proof*⟩

definition *valid-dbm* **where** *valid-dbm M n* ≡ *dbm-int M n* ∧ (∀ $i \leq n$. *M*
0 i ≤ **1**)

lemma *dbm-positive*:
assumes *M 0* (*v c*) ≤ **1** *v c* ≤ *n DBM-val-bounded v u M n*
shows *u c* ≥ *0*
⟨*proof*⟩

lemma *valid-dbm-pos*:
assumes *valid-dbm M n*
shows $[M]_{v,n} \subseteq \{u.\ \forall\ c.\ v\ c \leq n \longrightarrow u\ c \geq 0\}$
⟨*proof*⟩

lemma (**in** *Regions'*) *V-alt-def*:
shows $\{u.\ \forall\ c.\ v\ c > 0 \wedge v\ c \leq n \longrightarrow u\ c \geq 0\} = V$
⟨*proof*⟩

An example of obtaining concrete models from our formalizations.

lemma *steps-z-norm-sound-spec*:
assumes *finite-ta A*
obtains *k v n* **where**

$A \vdash \langle l,D\rangle \rightsquigarrow_{k,v,n}* \langle l',D'\rangle \wedge \mathit{valid\text{-}dbm}\ D\ n \wedge [D']_{v,n} \neq \{\}$
$\longrightarrow (\exists\, Z.\ A \vdash \langle l,\ [D]_{v,n}\rangle \rightsquigarrow* \langle l',Z\rangle \wedge Z \neq \{\})$
$\langle proof \rangle$

end

References

[AD90] Rajeev Alur and D. L. Dill. Automata for modeling real-time systems. In *Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming*, pages 322–335, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[Bou04] Patricia Bouyer. Forward analysis of updatable timed automata. *Formal Methods in System Design*, 24(3):281–320, 2004.

[BY03] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets, Advances in Petri Nets [This tutorial volume originates from the 4th Advanced Course on Petri Nets, ACPN 2003, held in Eichstätt, Germany in September 2003. In addition to lectures given at ACPN 2003, additional chapters have been commissioned]*, pages 87–124, 2003.

[HHWt97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-toi. Hytech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:460–463, 1997.

[LPY97] G. Kim Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1):134–152, 1997.

[Yov97] Sergio Yovine. KRONOS: A verification tool for real-time systems. *STTT*, 1(1-2):123–133, 1997.