Difference Bound Matrices

Simon Wimmer, Peter Lammich

March 17, 2025

Abstract

Difference Bound Matrices (DBMs) [2] are a data structure used to represent a type of convex polytopes, often called zones. DBMs find application such as in timed automata model checking and static program analysis. This entry formalizes DBMs and operations for inclusion checking, intersection, variable reset, upper-bound relaxation, and extrapolation (as used in timed automata model checking). With the help of the Imperative Refinement Framework, efficient imperative implementations of these operations are also provided. For each zone, there exists a canonical DBM. The characteristic properties of canonical forms are proved, including the fact that DBMs can be transformed in canonical form by the Floyd-Warshall algorithm. This entry is part of the work described in a paper by the authors of this entry [4] and a PhD thesis [3].

Contents

theory *DBM*
  imports
    Floyd-Warshall.Floyd-Warshall
    HOL.Real
begin

type-synonym $('c, 't)$ *cval* $= 'c \Rightarrow 't$

1 Difference Bound Matrices

1.1 Definitions

1.1.1 Definition and Semantics of DBMs

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 $'t$ *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 $'c \Rightarrow nat$ 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**.

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 −
  from *non-trivial* **obtain** *x* **where** *x*: $x \neq 0$ **by** *auto*
  show *?thesis*
  proof (*cases* $x < 0$)
    case *False*
    with *x* **have** $x > 0$ **by** *auto*
    then have $(-x) < 0$ **by** *auto*
    then show *?thesis* **..**

qed *auto*
qed

end

instantiation *real* :: *time*
begin
instance proof
fix *x y* :: *real*
assume $x < y$
then show $\exists z>x.\ z < y$ **using** *dense-order-class.dense* **by** *blast*
next
have $(1 :: real) \neq 0$ **by** *auto*
then show $\exists x.\ (x{::}real) \neq 0$ **..**
qed
end

inductive *dbm-entry-val* :: ('c, 't) *cval* ⇒ 'c *option* ⇒ 'c *option* ⇒ ('t::*time*) *DBMEntry* ⇒ *bool*
where
$u\ r \leq d \Longrightarrow$ *dbm-entry-val u* (*Some r*) *None* (*Le d*) |
$-u\ c \leq d \Longrightarrow$ *dbm-entry-val u None* (*Some c*) (*Le d*) |
$u\ r < d \Longrightarrow$ *dbm-entry-val u* (*Some r*) *None* (*Lt d*) |
$-u\ c < d \Longrightarrow$ *dbm-entry-val u None* (*Some c*) (*Lt d*) |
$u\ r - u\ c \leq d \Longrightarrow$ *dbm-entry-val u* (*Some r*) (*Some c*) (*Le d*) |
$u\ r - u\ c < d \Longrightarrow$ *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::*linorder*) *DBMEntry* ⇒ 't *DBMEntry* ⇒ *bool*

(‹- $\prec$ -› [*51*, *51*] *50*)
where
dbm-lt (*Lt* -) ∞ |
dbm-lt (*Le* -) ∞ |
$a < b \implies$ *dbm-lt* (*Le a*) (*Le b*) |
$a < b \implies$ *dbm-lt* (*Le a*) (*Lt b*) |
$a \leq b \implies$ *dbm-lt* (*Lt a*) (*Le b*) |
$a < b \implies$ *dbm-lt* (*Lt a*) (*Lt b*)

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

definition *dbm-le* :: (*'t*::*linorder*) *DBMEntry* ⇒ *'t DBMEntry* ⇒ *bool*
(‹- $\preceq$ -› [*51*, *51*] *50*)
where
dbm-le a b ≡ ($a \prec b$) ∨ $a = b$

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

definition *DBM-val-bounded* :: (*'c* ⇒ *nat*) ⇒ (*'c*, *'t*) *cval* ⇒ (*'t*::*time*) *DBM* ⇒ *nat* ⇒ *bool*
where
DBM-val-bounded v u m n ≡ *Le 0* $\preceq$ *m 0 0* ∧
(∀ *c*. *v c* ≤ *n* ⟶ (*dbm-entry-val u None* (*Some c*) (*m 0* (*v c*))
∧ *dbm-entry-val u* (*Some c*) *None* (*m* (*v c*) *0*)))
∧ (∀ *c1 c2*. *v c1* ≤ *n* ∧ *v c2* ≤ *n* ⟶ *dbm-entry-val u* (*Some c1*) (*Some c2*) (*m* (*v c1*) (*v c2*)))

abbreviation *DBM-val-bounded-abbrev* ::
(*'c*, *'t*) *cval* ⇒ (*'c* ⇒ *nat*) ⇒ *nat* ⇒ (*'t*::*time*) *DBM* ⇒ *bool*
(‹- $\vdash_{-,-}$ -› [*48*, *48*, *48*, *48*] *48*)
where
$u \vdash_{v,n} M$ ≡ *DBM-val-bounded v u M n*

1.1.2 Ordering DBM Entries

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

lemma *dbm-le-dbm-min*:
$a \preceq b \implies a =$ *dmin a b* **unfolding** *dbm-le-def*
by *auto*

lemma *dbm-lt-asym*:
assumes $e \prec f$

shows $\sim f \prec e$
using *assms*
proof (*safe*, *cases e f rule*: *dbm-lt.cases*, *goal-cases*)
case *1* **from** *this*(*2*) **show** *?case* **using** *1*(*3*−) **by** (*cases f e rule*: *dbm-lt.cases*) *auto*
next
case *2* **from** *this*(*2*) **show** *?case* **using** *2*(*3*−) **by** (*cases f e rule*: *dbm-lt.cases*) *auto*
next
case *3* **from** *this*(*2*) **show** *?case* **using** *3*(*3*−) **by** (*cases f e rule*: *dbm-lt.cases*) *auto*
next
case *4* **from** *this*(*2*) **show** *?case* **using** *4*(*3*−) **by** (*cases f e rule*: *dbm-lt.cases*) *auto*
next
case *5* **from** *this*(*2*) **show** *?case* **using** *5*(*3*−) **by** (*cases f e rule*: *dbm-lt.cases*) *auto*
next
case *6* **from** *this*(*2*) **show** *?case* **using** *6*(*3*−) **by** (*cases f e rule*: *dbm-lt.cases*) *auto*
qed

lemma *dbm-le-dbm-min2*:
$a \preceq b \Longrightarrow a = dmin\ b\ a$
using *dbm-lt-asym* **by** (*auto simp*: *dbm-le-def*)

lemma *dmb-le-dbm-entry-bound-inf*:
$a \preceq b \Longrightarrow a = \infty \Longrightarrow b = \infty$
by (*auto simp*: *dbm-le-def elim*: *dbm-lt.cases*)

lemma *dbm-not-lt-eq*: $\neg\ a \prec b \Longrightarrow \neg\ b \prec a \Longrightarrow a = b$
by (*cases a*; *cases b*; *fastforce*)

lemma *dbm-not-lt-impl*: $\neg\ a \prec b \Longrightarrow b \prec a \lor a = b$ **using** *dbm-not-lt-eq*
by *auto*

lemma $dmin\ a\ b = dmin\ b\ a$
proof (*cases* $a \prec b$)
case *True* **thus** *?thesis* **by** (*simp add*: *dbm-lt-asym*)
next
case *False* **thus** *?thesis* **by** (*simp add*: *dbm-not-lt-eq*)
qed

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

proof (*cases a b rule*: *dbm-lt.cases*, *goal-cases*)
  case *1* **thus** *?case* **by** *simp*
next
  case *2* **from** *this*(*2*−) **show** *?case* **by** (*cases rule*: *dbm-lt.cases*) *simp*+
next
  case *3* **from** *this*(*2*−) **show** *?case* **by** (*cases rule*: *dbm-lt.cases*) *simp*+
next
  case *4* **from** *this*(*2*−) **show** *?case* **by** (*cases rule*: *dbm-lt.cases*) *auto*
next
  case *5* **from** *this*(*2*−) **show** *?case* **by** (*cases rule*: *dbm-lt.cases*) *auto*
next
  case *6* **from** *this*(*2*−) **show** *?case* **by** (*cases rule*: *dbm-lt.cases*) *auto*
next
  case *7* **from** *this*(*2*−) **show** *?case* **by** (*cases rule*: *dbm-lt.cases*) *auto*
qed

lemma *aux-3*: $\neg\ a \prec b \Longrightarrow \neg\ b \prec c \Longrightarrow a \prec c \Longrightarrow c = a$
proof *goal-cases*
  case *1* **thus** *?case*
  proof (*cases* $c \prec b$)
    case *True*
    with ‹$a \prec c$› **have** $a \prec b$ **by** (*rule dbm-lt-trans*)
    thus *?thesis* **using** *1* **by** *auto*
  next
    case *False* **thus** *?thesis* **using** *dbm-not-lt-eq 1* **by** *auto*
  qed
qed

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

lemma *dbm-lt-asymmetric*[*simp*]: $x \prec y \Longrightarrow y \prec x \Longrightarrow$ *False*
by (*cases x y rule*: *dbm-lt.cases*) (*auto elim*: *dbm-lt.cases*)

lemma *le-dbm-le*: *Le* $a \preceq$ *Le* $b \Longrightarrow a \le b$ **unfolding** *dbm-le-def* **by** (*auto elim*: *dbm-lt.cases*)

lemma *le-dbm-lt*: *Le* $a \preceq$ *Lt* $b \Longrightarrow a < b$ **unfolding** *dbm-le-def* **by** (*auto elim*: *dbm-lt.cases*)

lemma *lt-dbm-le*: *Lt* $a \preceq$ *Le* $b \Longrightarrow a \le b$ **unfolding** *dbm-le-def* **by** (*auto elim*: *dbm-lt.cases*)

lemma *lt-dbm-lt*: *Lt* $a \preceq$ *Lt* $b \Longrightarrow a \le b$ **unfolding** *dbm-le-def* **by** (*auto elim*: *dbm-lt.cases*)

lemma *not-dbm-le-le-impl*: $\neg$ *Le a* $\prec$ *Le b* $\Longrightarrow$ $a \geq b$ **by** (*metis dbm-lt.intros*(*3*) *not-less*)

lemma *not-dbm-lt-le-impl*: $\neg$ *Lt a* $\prec$ *Le b* $\Longrightarrow$ $a > b$ **by** (*metis dbm-lt.intros*(*5*) *not-less*)

lemma *not-dbm-lt-lt-impl*: $\neg$ *Lt a* $\prec$ *Lt b* $\Longrightarrow$ $a \geq b$ **by** (*metis dbm-lt.intros*(*6*) *not-less*)

lemma *not-dbm-le-lt-impl*: $\neg$ *Le a* $\prec$ *Lt b* $\Longrightarrow$ $a \geq b$ **by** (*metis dbm-lt.intros*(*4*) *not-less*)

1.1.3 Addition on DBM Entries

fun *dbm-add* :: (*'t::linordered-cancel-ab-semigroup-add*) *DBMEntry* $\Rightarrow$ *'t DBMEntry* $\Rightarrow$ *'t DBMEntry* (**infixl** ‹⊗› *70*)
where
  dbm-add ∞ - = ∞ |
  dbm-add - ∞ = ∞ |
  dbm-add (*Le a*) (*Le b*) = (*Le* (*a*+*b*)) |
  dbm-add (*Le a*) (*Lt b*) = (*Lt* (*a*+*b*)) |
  dbm-add (*Lt a*) (*Le b*) = (*Lt* (*a*+*b*)) |
  dbm-add (*Lt a*) (*Lt b*) = (*Lt* (*a*+*b*))

lemma *aux-4*: $x \prec y$ $\Longrightarrow$ $\neg$ *dbm-add x z* $\prec$ *dbm-add y z* $\Longrightarrow$ *dbm-add x z* = *dbm-add y z*
by (*cases x y rule*: *dbm-lt.cases*; *cases z*; *auto*)

lemma *aux-5*: $\neg$ $x \prec y$ $\Longrightarrow$ *dbm-add x z* $\prec$ *dbm-add y z* $\Longrightarrow$ *dbm-add y z* = *dbm-add x z*
proof −
  assume *lt*: *dbm-add x z* $\prec$ *dbm-add y z* $\neg$ $x \prec y$
  hence $x = y \vee y \prec x$ **by** (*auto simp*: *dbm-not-lt-eq*)
  thus *?thesis*
  proof
    assume $x = y$ **thus** *?thesis* **by** *simp*
  next
    assume $y \prec x$
    thus *?thesis*
    proof (*cases y x rule*: *dbm-lt.cases*, *goal-cases*)
      case *1* **thus** *?case* **using** *lt* **by** *auto*
    next
      case *2* **thus** *?case* **using** *lt* **by** *auto*

next
case *3* **thus** *?case* **using** *dbm-lt-asymmetric lt*(*1*) **by** (*cases z*) *fastforce+*
next
case *4* **thus** *?case* **using** *dbm-lt-asymmetric lt*(*1*) **by** (*cases z*) *fastforce+*
next
case *5* **thus** *?case* **using** *dbm-lt-asymmetric lt*(*1*) **by** (*cases z*) *fastforce+*
next
case *6* **thus** *?case* **using** *dbm-lt-asymmetric lt*(*1*) **by** (*cases z*) *fastforce+*
qed
qed
qed

lemma *aux-42*: $x \prec y \Longrightarrow \neg$ *dbm-add z x* $\prec$ *dbm-add z y* $\Longrightarrow$ *dbm-add z x* = *dbm-add z y*
by (*cases x y rule*: *dbm-lt.cases*) ((*cases z*), *auto*)+

lemma *aux-52*: $\neg\ x \prec y \Longrightarrow$ *dbm-add z x* $\prec$ *dbm-add z y* $\Longrightarrow$ *dbm-add z y* = *dbm-add z x*
proof −
assume *lt*: *dbm-add z x* $\prec$ *dbm-add z y* $\neg\ x \prec y$
hence $x = y \lor y \prec x$ **by** (*auto simp*: *dbm-not-lt-eq*)
thus *?thesis*
proof
assume $x = y$ **thus** *?thesis* **by** *simp*
next
assume $y \prec x$
thus *?thesis*
proof (*cases y x rule*: *dbm-lt.cases*, *goal-cases*)
case *1* **thus** *?case* **using** *lt* **by** (*cases z*) *fastforce+*
next
case *2* **thus** *?case* **using** *lt* **by** (*cases z*) *fastforce+*
next
case *3* **thus** *?case* **using** *dbm-lt-asymmetric lt*(*1*) **by** (*cases z*) *fastforce+*
next
case *4* **thus** *?case* **using** *dbm-lt-asymmetric lt*(*1*) **by** (*cases z*) *fastforce+*
next
case *5* **thus** *?case* **using** *dbm-lt-asymmetric lt*(*1*) **by** (*cases z*) *fastforce+*

next
case *6* **thus** *?case* **using** *dbm-lt-asymmetric lt*(*1*) **by** (*cases z*) *fast-force+*
qed
qed
qed

lemma *dbm-add-not-inf*:
$a \neq \infty \Longrightarrow b \neq \infty \Longrightarrow$ *dbm-add a b* $\neq \infty$
by (*cases a*; *cases b*; *auto*)

lemma *dbm-le-not-inf*:
$a \preceq b \Longrightarrow b \neq \infty \Longrightarrow a \neq \infty$
by (*cases a* = *b*) (*auto simp*: *dbm-le-def*)

1.1.4 Negation of DBM Entries

fun *neg-dbm-entry* **where**
neg-dbm-entry (*Le a*) = *Lt* ($-a$) |
neg-dbm-entry (*Lt a*) = *Le* ($-a$) |
neg-dbm-entry $\infty = \infty$
— This case does not make sense but we make this definition for technical convenience.

lemma *neg-entry*:
{*u*. ¬ *dbm-entry-val u a b e*} = {*u*. *dbm-entry-val u b a* (*neg-dbm-entry e*)}
if $e \neq (\infty$ :: - *DBMEntry*) $a \neq$ *None* ∨ $b \neq$ *None*
using *that* **by** (*cases e*; *cases a*; *cases b*; *auto 4 3 simp*: *le-minus-iff less-minus-iff*)

instantiation *DBMEntry* :: (*uminus*) *uminus*
begin
definition *uminus*: *uminus* = *neg-dbm-entry*
instance ..
end

Note that it is not clear that this is the only sensible definition for negation of DBM entries. The following would also have been quite viable: *fun neg-dbm-entry where neg-dbm-entry* (*Le a*) = *Le* ($-a$) | *neg-dbm-entry* (*Lt a*) = *Lt* ($-a$) | *neg-dbm-entry* $\infty = \infty$

For most practical proofs using arithmetic on DBM entries we have found that this does not make much of a difference. Lemma ⟦*?e* $\neq \infty$; *?a* $\neq$ *None* ∨ *?b* $\neq$ *None*⟧ $\Longrightarrow$ {*u*. ¬ *dbm-entry-val u ?a ?b ?e*} = {*u*. *dbm-entry-val u*

?b ?a (*neg-dbm-entry ?e*)} would not hold any longer, however.

1.2 DBM Entries Form a Linearly Ordered Abelian Monoid

instantiation *DBMEntry* :: (*linorder*) *linorder*
begin
  definition *less-eq*: (≤) ≡ *dbm-le*
  definition *less*: (<) = *dbm-lt*
  instance
  proof ((*standard*; *unfold less less-eq*), *goal-cases*)
    case *1* **thus** *?case* **unfolding** *dbm-le-def* **using** *dbm-lt-asymmetric* **by** *auto*
  next
    case *2* **thus** *?case* **by** (*simp add*: *dbm-le-def*)
  next
    case *3* **thus** *?case* **unfolding** *dbm-le-def* **using** *dbm-lt-trans* **by** *auto*
  next
    case *4* **thus** *?case* **unfolding** *dbm-le-def* **using** *dbm-lt-asymmetric* **by** *auto*
  next
    case *5* **thus** *?case* **unfolding** *dbm-le-def* **using** *dbm-not-lt-eq* **by** *auto*
  qed
end

class *linordered-cancel-ab-monoid-add* =
  linordered-cancel-ab-semigroup-add + *zero* +
    assumes *neutl*[*simp*]: *0* + *x* = *x*
    assumes *neutr*[*simp*]: *x* + *0* = *x*
begin

  subclass *linordered-ab-monoid-add*
    by *standard* (*rule neutl*)

end

instantiation *DBMEntry* :: (*zero*) *zero*
begin
  definition *neutral*: *0* = *Le 0*
  instance ..
end

instantiation *DBMEntry* :: (*linordered-cancel-ab-monoid-add*) *linordered-ab-monoid-add*
begin

definition *add*: (+) = *dbm-add*

instance proof ((*standard*; *unfold add neutral less less-eq*), *goal-cases*)
  case (*1 a b c*) **thus** *?case* **by** (*cases a*; *cases b*; *cases c*; *auto simp*: *add.assoc*)
next
  case (*2 a b*) **thus** *?case* **by** (*cases a*; *cases b*; *auto simp*: *add.commute*)
next
  case (*3 a*) **thus** *?case* **by** (*cases a*) *auto*
next
  case (*4 a b c*)
  thus *?case* **unfolding** *dbm-le-def*
  apply *safe*
  apply (*rule dbm-lt.cases*)
  apply *assumption*
  by (*cases c*; *fastforce*)+
qed

end

interpretation *linordered-monoid*:
linordered-ab-monoid-add dbm-add Le (*0*::*'t*::*linordered-cancel-ab-monoid-add*) *dbm-le dbm-lt*
apply (*standard*, *fold neutral add less-eq less*)
using *add.commute* **by** (*auto intro*: *add-left-mono simp*: *add.assoc*)

instance *time* ⊆ *linordered-cancel-ab-monoid-add* **by** (*standard*; *simp*)

lemma *dbm-add-strict-right-mono-neutral*: $a < Le\ (d :: 't :: time) \Longrightarrow a + Le\ (-d) < Le\ 0$
unfolding *less add* **by** (*cases a*) (*auto elim*!: *dbm-lt.cases*)

lemma *dbm-lt-not-inf-less*[*intro*]: $A \neq \infty \Longrightarrow A \prec \infty$ **by** (*cases A*) *auto*

lemma *add-inf*[*simp*]:
$a + \infty = \infty$ $\infty + a = \infty$
unfolding *add* **by** (*cases a*) *auto*

lemma *inf-lt*[*simp*,*dest*!]:
$\infty < x \Longrightarrow False$
by (*cases x*) (*auto simp*: *less*)

lemma *inf-lt-impl-False*[*simp*]:
$\infty < x = False$

by *auto*

lemma *Le-Le-dbm-lt-D*[*dest*]: *Le a* ≺ *Lt b* ⟹ *a* < *b* **by** (*cases rule*: *dbm-lt.cases*) *auto*
lemma *Le-Lt-dbm-lt-D*[*dest*]: *Le a* ≺ *Le b* ⟹ *a* < *b* **by** (*cases rule*: *dbm-lt.cases*) *auto*
lemma *Lt-Le-dbm-lt-D*[*dest*]: *Lt a* ≺ *Le b* ⟹ *a* ≤ *b* **by** (*cases rule*: *dbm-lt.cases*) *auto*
lemma *Lt-Lt-dbm-lt-D*[*dest*]: *Lt a* ≺ *Lt b* ⟹ *a* < *b* **by** (*cases rule*: *dbm-lt.cases*) *auto*

lemma *Le-le-LeI*[*intro*]: *a* ≤ *b* ⟹ *Le a* ≤ *Le b* **unfolding** *less-eq dbm-le-def* **by** *auto*
lemma *Lt-le-LeI*[*intro*]: *a* ≤ *b* ⟹ *Lt a* ≤ *Le b* **unfolding** *less-eq dbm-le-def* **by** *auto*
lemma *Lt-le-LtI*[*intro*]: *a* ≤ *b* ⟹ *Lt a* ≤ *Lt b* **unfolding** *less-eq dbm-le-def* **by** *auto*
lemma *Le-le-LtI*[*intro*]: *a* < *b* ⟹ *Le a* ≤ *Lt b* **unfolding** *less-eq dbm-le-def* **by** *auto*
lemma *Lt-lt-LeI*: *x* ≤ *y* ⟹ *Lt x* < *Le y* **unfolding** *less* **by** *auto*

lemma *Le-le-LeD*[*dest*]: *Le a* ≤ *Le b* ⟹ *a* ≤ *b* **unfolding** *dbm-le-def less-eq* **by** *auto*
lemma *Le-le-LtD*[*dest*]: *Le a* ≤ *Lt b* ⟹ *a* < *b* **unfolding** *dbm-le-def less-eq* **by** *auto*
lemma *Lt-le-LeD*[*dest*]: *Lt a* ≤ *Le b* ⟹ *a* ≤ *b* **unfolding** *less-eq dbm-le-def* **by** *auto*
lemma *Lt-le-LtD*[*dest*]: *Lt a* ≤ *Lt b* ⟹ *a* ≤ *b* **unfolding** *less-eq dbm-le-def* **by** *auto*

lemma *inf-not-le-Le*[*simp*]: ∞ ≤ *Le x* = *False* **unfolding** *less-eq dbm-le-def* **by** *auto*
lemma *inf-not-le-Lt*[*simp*]: ∞ ≤ *Lt x* = *False* **unfolding** *less-eq dbm-le-def* **by** *auto*
lemma *inf-not-lt*[*simp*]: ∞ ≺ *x* = *False* **by** *auto*

lemma *any-le-inf*: *x* ≤ (∞ :: - *DBMEntry*) **by** (*metis less-eq dmb-le-dbm-entry-bound-inf le-cases*)

lemma *dbm-lt-code-simps*[*code*]:
dbm-lt (*Lt a*) ∞ = *True*
dbm-lt (*Le a*) ∞ = *True*
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*) = (*a* < *b*)
dbm-lt ∞ *x* = *False*
by *auto*

1.3 Basic Properties of DBMs

1.3.1 DBMs and Length of Paths

lemma *dbm-entry-val-add-1*: *dbm-entry-val u* (*Some c*) (*Some d*) *a* ⟹ *dbm-entry-val u* (*Some d*) *None b*
⟹ *dbm-entry-val u* (*Some c*) *None* (*dbm-add a b*)
proof (*cases a*, *goal-cases*)
case *1* **thus** *?thesis*
apply (*cases b*)
using *add-mono-thms-linordered-semiring*(*1*) *add-le-less-mono* **by** *auto fastforce+*
next
case *2* **thus** *?thesis*
apply (*cases b*)
apply (*clarsimp simp*: *dbm-entry-val.intros*(*3*) *diff-less-eq less-le-trans*)
apply (*clarsimp*, *metis add-le-less-mono dbm-entry-val.intros*(*3*) *diff-add-cancel less-imp-le*)
apply *auto*
done
next
case *3* **thus** *?thesis* **by** (*cases b*) *auto*
qed

lemma *dbm-entry-val-add-2*: *dbm-entry-val u None* (*Some c*) *a* ⟹ *dbm-entry-val u* (*Some c*) (*Some d*) *b*
⟹ *dbm-entry-val u None* (*Some d*) (*dbm-add a b*)
proof (*cases a*, *goal-cases*)
case *1* **thus** *?thesis*
apply (*cases b*)
using *add-mono-thms-linordered-semiring*(*1*) *add-le-less-mono* **by** *fastforce+*
next
case *2* **thus** *?thesis*
apply (*cases b*)
using *add-mono-thms-linordered-field*(*3*) **apply** *fastforce*
using *add-strict-mono* **by** *fastforce+*
next
case *3* **thus** *?thesis* **by** (*cases b*) *auto*

qed

lemma *dbm-entry-val-add-3*:
dbm-entry-val u (*Some c*) (*Some d*) *a* ⟹ *dbm-entry-val u* (*Some d*) (*Some e*) *b*
⟹ *dbm-entry-val u* (*Some c*) (*Some e*) (*dbm-add a b*)
proof (*cases a*, *goal-cases*)
case *1* **thus** *?thesis*
apply (*cases b*)
using *add-mono-thms-linordered-semiring*(*1*) **apply** *fastforce*
using *add-le-less-mono* **by** *fastforce+*
next
case *2* **thus** *?thesis*
apply (*cases b*)
using *add-mono-thms-linordered-field*(*3*) **apply** *fastforce*
using *add-strict-mono* **by** *fastforce+*
next
case *3* **thus** *?thesis* **by** (*cases b*) *auto*
qed

lemma *dbm-entry-val-add-4*:
dbm-entry-val u (*Some c*) *None a* ⟹ *dbm-entry-val u None* (*Some d*) *b*
⟹ *dbm-entry-val u* (*Some c*) (*Some d*) (*dbm-add a b*)
proof (*cases a*, *goal-cases*)
case *1* **thus** *?thesis*
apply (*cases b*)
using *add-mono-thms-linordered-semiring*(*1*) **apply** *fastforce*
using *add-le-less-mono* **by** *fastforce+*
next
case *2* **thus** *?thesis*
apply (*cases b*)
using *add-mono-thms-linordered-field*(*3*) **apply** *fastforce*
using *add-strict-mono* **by** *fastforce+*
next
case *3* **thus** *?thesis* **by** (*cases b*) *auto*
qed

no-notation *dbm-add* (**infixl** ‹⊗› *70*)

lemma *DBM-val-bounded-len-1'-aux*:
assumes *DBM-val-bounded v u m n v c* ≤ *n* ∀ *k* ∈ *set vs*. *k* > *0* ∧ *k* ≤ *n* ∧ (∃ *c*. *v c* = *k*)
shows *dbm-entry-val u* (*Some c*) *None* (*len m* (*v c*) *0 vs*) **using** *assms*
proof (*induction vs arbitrary*: *c*)

case *Nil* **then show** *?case* **unfolding** *DBM-val-bounded-def* **by** *auto*
next
case (*Cons k vs*)
then obtain c' **where** c': $k > 0$ $k \leq n$ $v\ c' = k$ **by** *auto*
with *Cons* **have** *dbm-entry-val u* (*Some* c') *None* (*len m* ($v\ c'$) *0 vs*) **by** *auto*
moreover have *dbm-entry-val u* (*Some c*) (*Some* c') (*m* (*v c*) ($v\ c'$)) **using** *Cons.prems* c'
by (*auto simp add*: *DBM-val-bounded-def*)
ultimately have *dbm-entry-val u* (*Some c*) *None* (*m* (*v c*) ($v\ c'$) + *len m* ($v\ c'$) *0 vs*)
using *dbm-entry-val-add-1* **unfolding** *add* **by** *fastforce*
with c' **show** *?case* **unfolding** *DBM-val-bounded-def* **by** *simp*
qed

lemma *DBM-val-bounded-len-3'-aux*:
DBM-val-bounded v u m n $\Longrightarrow$ $v\ c \leq n$ $\Longrightarrow$ $v\ d \leq n$ $\Longrightarrow$ $\forall\ k \in set\ vs.\ k > 0 \wedge k \leq n \wedge (\exists\ c.\ v\ c = k)$
$\Longrightarrow$ *dbm-entry-val u* (*Some c*) (*Some d*) (*len m* (*v c*) (*v d*) *vs*)
proof (*induction vs arbitrary*: *c*)
case *Nil* **thus** *?case* **unfolding** *DBM-val-bounded-def* **by** *auto*
next
case (*Cons k vs*)
then obtain c' **where** c': $k > 0$ $k \leq n$ $v\ c' = k$ **by** *auto*
with *Cons* **have** *dbm-entry-val u* (*Some* c') (*Some d*) (*len m* ($v\ c'$) (*v d*) *vs*) **by** *auto*
moreover have *dbm-entry-val u* (*Some c*) (*Some* c') (*m* (*v c*) ($v\ c'$)) **using** *Cons.prems* c'
by (*auto simp add*: *DBM-val-bounded-def*)
ultimately have *dbm-entry-val u* (*Some c*) (*Some d*) (*m* (*v c*) ($v\ c'$) + *len m* ($v\ c'$) (*v d*) *vs*)
using *dbm-entry-val-add-3* **unfolding** *add* **by** *fastforce*
with c' **show** *?case* **unfolding** *DBM-val-bounded-def* **by** *simp*
qed

lemma *DBM-val-bounded-len-2'-aux*:
DBM-val-bounded v u m n $\Longrightarrow$ $v\ c \leq n$ $\Longrightarrow$ $\forall\ k \in set\ vs.\ k > 0 \wedge k \leq n \wedge (\exists\ c.\ v\ c = k)$
$\Longrightarrow$ *dbm-entry-val u None* (*Some c*) (*len m 0* (*v c*) *vs*)
proof (*cases vs*, *goal-cases*)
case *1* **then show** *?thesis* **unfolding** *DBM-val-bounded-def* **by** *auto*
next
case (*2 k vs*)
then obtain c' **where** c': $k > 0$ $k \leq n$ $v\ c' = k$ **by** *auto*

with *2* **have** *dbm-entry-val u* (*Some c'*) (*Some c*) (*len m* (*v c'*) (*v c*) *vs*)
using *DBM-val-bounded-len-3'-aux* **by** *auto*
moreover have *dbm-entry-val u None* (*Some c'*) (*m 0* (*v c'*))
using *2 c'* **by** (*auto simp add*: *DBM-val-bounded-def*)
ultimately have *dbm-entry-val u None* (*Some c*) (*m 0* (*v c'*) + *len m* (*v c'*) (*v c*) *vs*)
using *dbm-entry-val-add-2* **unfolding** *add* **by** *fastforce*
with *2*(*4*) *c'* **show** *?case* **unfolding** *DBM-val-bounded-def* **by** *simp*
qed

lemma *cnt-0-D*:
cnt x xs = *0* ⟹ *x* ∉ *set xs*
apply (*induction xs*)
apply *simp*
subgoal for *a xs*
by (*cases x* = *a*; *simp*)
done

lemma *cnt-at-most-1-D*:
cnt x (*xs* @ *x* # *ys*) ≤ *1* ⟹ *x* ∉ *set xs* ∧ *x* ∉ *set ys*
apply (*induction xs*)
apply *auto*[]
using *cnt-0-D* **apply** *force*
subgoal for *a xs*
by (*cases x* = *a*; *simp*)
done

lemma *nat-list-0* [*intro*]:
x ∈ *set xs* ⟹ *0* ∉ *set* (*xs* :: *nat list*) ⟹ *x* > *0*
by (*induction xs*) *auto*

lemma *DBM-val-bounded-len'1*:
fixes *v*
assumes *DBM-val-bounded v u m n 0* ∉ *set vs v c* ≤ *n*
∀ *k* ∈ *set vs*. *k* > *0* ⟶ *k* ≤ *n* ∧ (∃ *c*. *v c* = *k*)
shows *dbm-entry-val u* (*Some c*) *None* (*len m* (*v c*) *0 vs*)
using *DBM-val-bounded-len-1'-aux*[*OF assms*(*1*,*3*)] *assms*(*2*,*4*) **by** *fastforce*

lemma *DBM-val-bounded-len'2*:
fixes *v*
assumes *DBM-val-bounded v u m n 0* ∉ *set vs v c* ≤ *n*
∀ *k* ∈ *set vs*. *k* > *0* ⟶ *k* ≤ *n* ∧ (∃ *c*. *v c* = *k*)
shows *dbm-entry-val u None* (*Some c*) (*len m 0* (*v c*) *vs*)

using *DBM-val-bounded-len-2′-aux*[*OF assms*(*1*,*3*)] *assms*(*2*,*4*) **by** *fast-force*

lemma *DBM-val-bounded-len′3*:
fixes *v*
assumes *DBM-val-bounded v u m n cnt 0 vs* $\leq$ *1 v c1* $\leq$ *n v c2* $\leq$ *n*
$\forall$ *k* $\in$ *set vs. k* $>$ *0* $\longrightarrow$ *k* $\leq$ *n* $\wedge$ ($\exists$ *c. v c* = *k*)
shows *dbm-entry-val u* (*Some c1*) (*Some c2*) (*len m* (*v c1*) (*v c2*) *vs*)
proof −
show *?thesis*
proof (*cases* $\forall$ *k* $\in$ *set vs. k* $>$ *0*)
case *True*
with *assms* **have** $\forall$ *k* $\in$ *set vs. k* $>$ *0* $\wedge$ *k* $\leq$ *n* $\wedge$ ($\exists$ *c. v c* = *k*) **by** *auto*
with *DBM-val-bounded-len-3′-aux*[*OF assms*(*1*,*3*,*4*)] **show** *?thesis* **by** *auto*
next
case *False*
then have $\exists$ *k* $\in$ *set vs. k* = *0* **by** *auto*
then obtain *us ws* **where** *vs*: *vs* = *us* @ *0* # *ws* **by** (*meson split-list-last*)
with *cnt-at-most-1-D*[*of 0 us*] *assms*(*2*) **have**
0 $\notin$ *set us 0* $\notin$ *set ws*
by *auto*
with *vs* **have** *vs*: *vs* = *us* @ *0* # *ws* $\forall$ *k* $\in$ *set us. k* $>$ *0* $\forall$ *k* $\in$ *set ws. k* $>$ *0* **by** *auto*
with *assms*(*5*) **have** *v*:
$\forall$*k*$\in$*set us. 0* $<$ *k* $\wedge$ *k* $\leq$ *n* $\wedge$ ($\exists$*c. v c* = *k*) $\forall$*k*$\in$*set ws. 0* $<$ *k* $\wedge$ *k* $\leq$ *n* $\wedge$ ($\exists$*c. v c* = *k*)
by *auto*
with
dbm-entry-val-add-4[*OF*
DBM-val-bounded-len-1′-aux[*OF assms*(*1*,*3*) *v*(*1*)]
DBM-val-bounded-len-2′-aux[*OF assms*(*1*,*4*) *v*(*2*)]
]
have *dbm-entry-val u* (*Some c1*) (*Some c2*) (*dbm-add* (*len m* (*v c1*) *0 us*) (*len m 0* (*v c2*) *ws*))
by *auto*
moreover from *vs* **have** *len m* (*v c1*) (*v c2*) *vs* = *dbm-add* (*len m* (*v c1*) *0 us*) (*len m 0* (*v c2*) *ws*)
by (*simp add*: *len-comp add*)
ultimately show *?thesis* **by** *auto*
qed
qed

```
Now unused   lemma DBM-val-bounded-len':
  fixes v
  defines vo ≡ λ k. if k = 0 then None else Some (SOME c. v c = k)
  assumes DBM-val-bounded v u m n cnt 0 (i # j # vs) ≤ 1
        ∀ k ∈ set (i # j # vs). k > 0 ⟶ k ≤ n ∧ (∃ c. v c = k)
  shows dbm-entry-val u (vo i) (vo j) (len m i j vs)
proof –
  show ?thesis
  proof (cases ∀ k ∈ set vs. k > 0)
    case True
    with assms have *: ∀ k ∈ set vs. k > 0 ∧ k ≤ n ∧ (∃ c. v c = k) by
auto
    show ?thesis
    proof (cases i = 0)
      case True
      then have i: vo i = None by (simp add: vo-def)
      show ?thesis
      proof (cases j = 0)
        case True with assms ‹i = 0› show ?thesis by auto
      next
        case False
        with assms obtain c2 where c2: j ≤ n v c2 = j vo j = Some c2
        unfolding vo-def by (fastforce intro: someI)
        with ‹i = 0› i DBM-val-bounded-len-2'-aux[OF assms(2) - *] show
?thesis by auto
      qed
    next
      case False
      with assms(4) obtain c1 where c1: i ≤ n v c1 = i vo i = Some c1
      unfolding vo-def by (fastforce intro: someI)
      show ?thesis
      proof (cases j = 0)
        case True
        with DBM-val-bounded-len-1'-aux[OF assms(2) - *] c1 show ?thesis
by (auto simp: vo-def)
      next
        case False
        with assms obtain c2 where c2: j ≤ n v c2 = j vo j = Some c2
        unfolding vo-def by (fastforce intro: someI)
        with c1 DBM-val-bounded-len-3'-aux[OF assms(2) - - *] show ?thesis
by auto
      qed
    qed
  next
```

case *False*
then have ∃ *k* ∈ *set vs. k* = *0* **by** *auto*
then obtain *us ws* **where** *vs*: *vs* = *us* @ *0* # *ws* **by** (*meson split-list-last*)
with *cnt-at-most-1-D*[*of 0 i* # *j* # *us ws*] *assms*(*3*) **have**
0 ∉ *set us 0* ∉ *set ws i* ≠ *0 j* ≠ *0*
by *auto*
with *vs* **have** *vs*: *vs* = *us* @ *0* # *ws* ∀ *k* ∈ *set us. k* > *0* ∀ *k* ∈ *set ws. k* > *0* **by** *auto*
with *assms*(*4*) **have** *v*:
∀ *k*∈*set us. 0* < *k* ∧ *k* ≤ *n* ∧ (∃ *c. v c* = *k*) ∀ *k*∈*set ws. 0* < *k* ∧ *k* ≤ *n* ∧ (∃ *c. v c* = *k*)
by *auto*
from ‹*i* ≠ *0*› ‹*j* ≠ *0*› *assms* **obtain** *c1 c2* **where**
c1: *i* ≤ *n v c1* = *i vo i* = *Some c1* **and** *c2*: *j* ≤ *n v c2* = *j vo j* = *Some c2*
unfolding *vo-def* **by** (*fastforce intro*: *someI*)
with *dbm-entry-val-add-4* [*OF DBM-val-bounded-len-1'-aux*[*OF assms*(*2*) - *v*(*1*)] *DBM-val-bounded-len-2'-aux*[*OF assms*(*2*) - *v*(*2*)]]
have *dbm-entry-val u* (*Some c1*) (*Some c2*) (*dbm-add* (*len m* (*v c1*) *0 us*) (*len m 0* (*v c2*) *ws*)) **by** *auto*
moreover from *vs* **have** *len m* (*v c1*) (*v c2*) *vs* = *dbm-add* (*len m* (*v c1*) *0 us*) (*len m 0* (*v c2*) *ws*)
by (*simp add*: *len-comp add*)
ultimately show *?thesis* **using** *c1 c2* **by** *auto*
qed
qed

lemma *DBM-val-bounded-len-1*: *DBM-val-bounded v u m n* ⟹ *v c* ≤ *n* ⟹ ∀ *c* ∈ *set cs. v c* ≤ *n*
⟹ *dbm-entry-val u* (*Some c*) *None* (*len m* (*v c*) *0* (*map v cs*))
proof (*induction cs arbitrary*: *c*)
case *Nil* **thus** *?case* **unfolding** *DBM-val-bounded-def* **by** *auto*
next
case (*Cons c' cs*)
hence *dbm-entry-val u* (*Some c'*) *None* (*len m* (*v c'*) *0* (*map v cs*)) **by** *auto*
moreover have *dbm-entry-val u* (*Some c*) (*Some c'*) (*m* (*v c*) (*v c'*)) **using** *Cons.prems*
by (*simp add*: *DBM-val-bounded-def*)
ultimately have *dbm-entry-val u* (*Some c*) *None* (*m* (*v c*) (*v c'*) + *len m* (*v c'*) *0* (*map v cs*))
using *dbm-entry-val-add-1* **unfolding** *add* **by** *fastforce*
thus *?case* **unfolding** *DBM-val-bounded-def* **by** *simp*
qed

lemma *DBM-val-bounded-len-3*: *DBM-val-bounded v u m n* $\Longrightarrow$ *v c* $\leq$ *n* $\Longrightarrow$ *v d* $\leq$ *n* $\Longrightarrow$ $\forall$ *c* $\in$ *set cs. v c* $\leq$ *n*
$\Longrightarrow$ *dbm-entry-val u* (*Some c*) (*Some d*) (*len m* (*v c*) (*v d*) (*map v cs*))
proof (*induction cs arbitrary*: *c*)
case *Nil* **thus** *?case* **unfolding** *DBM-val-bounded-def* **by** *auto*
next
case (*Cons c′ cs*)
hence *dbm-entry-val u* (*Some c′*) (*Some d*) (*len m* (*v c′*) (*v d*) (*map v cs*)) **by** *auto*
moreover have *dbm-entry-val u* (*Some c*) (*Some c′*) (*m* (*v c*) (*v c′*)) **using** *Cons.prems*
by (*simp add*: *DBM-val-bounded-def*)
ultimately have *dbm-entry-val u* (*Some c*) (*Some d*) (*m* (*v c*) (*v c′*) + *len m* (*v c′*) (*v d*) (*map v cs*))
using *dbm-entry-val-add-3* **unfolding** *add* **by** *fastforce*
thus *?case* **unfolding** *DBM-val-bounded-def* **by** *simp*
qed

lemma *DBM-val-bounded-len-2*: *DBM-val-bounded v u m n* $\Longrightarrow$ *v c* $\leq$ *n* $\Longrightarrow$ $\forall$ *c* $\in$ *set cs. v c* $\leq$ *n*
$\Longrightarrow$ *dbm-entry-val u None* (*Some c*) (*len m 0* (*v c*) (*map v cs*))
proof (*cases cs*, *goal-cases*)
case *1* **thus** *?thesis* **unfolding** *DBM-val-bounded-def* **by** *auto*
next
case (*2 c′ cs*)
hence *dbm-entry-val u* (*Some c′*) (*Some c*) (*len m* (*v c′*) (*v c*) (*map v cs*))
using *DBM-val-bounded-len-3* **by** *auto*
moreover have *dbm-entry-val u None* (*Some c′*) (*m 0* (*v c′*))
using *2* **by** (*simp add*: *DBM-val-bounded-def*)
ultimately have *dbm-entry-val u None* (*Some c*) (*m 0* (*v c′*) + *len m* (*v c′*) (*v c*) (*map v cs*))
using *dbm-entry-val-add-2* **unfolding** *add* **by** *fastforce*
thus *?case* **using** *2*(*4*) **unfolding** *DBM-val-bounded-def* **by** *simp*
qed

lemmas *DBM-arith-defs* = *add neutral uminus*

end
theory *Paths-Cycles*
imports *Floyd-Warshall.Floyd-Warshall*
begin

2 Library for Paths, Arcs and Lengths

lemma *length-eq-distinct*:
  assumes *set xs = set ys distinct xs length xs = length ys*
  shows *distinct ys*
using *assms card-distinct distinct-card* **by** *fastforce*

2.1 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 (*induction xs*)
  case *Nil* **thus** *?case* **by** *auto*
next
  case (*Cons x xs*)
  then have *length xs* > *0* **by** *auto*
  then obtain *y ys* **where** *xs*: *xs* = *y* # *ys* **by** (*metis Suc-length-conv less-imp-Suc-add*)
  show *?case*
  proof (*cases* (*i*, *j*) = (*x*, *y*))
    case *True*
    with *xs* **have** *x* # *xs* = [] @ *i* # *j* # *ys* **by** *simp*
    then show *?thesis* **by** *auto*
  next
    case *False*
    then show *?thesis*
    proof (*cases length ys* > *0*, *goal-cases*)
      case *2*
      then have *ys* = [] **by** *auto*
      then have *arcs′* (*x*#*xs*) = {(*x*,*y*)} **using** *xs* **by** (*auto simp add*: *arcs′-def*)
      with *Cons.prems*(*2*) *2*(*1*) **show** *?case* **by** *auto*
    next
      case *True*
      with *xs Cons.prems*(*2*) *False* **have** (*i*, *j*) ∈ *arcs′ xs* **by** (*auto simp add*: *arcs′-def*)
      with *Cons.IH*[*OF* - *this*] *True xs* **obtain** *zs ys* **where** *xs* = *zs* @ *i* #

j # *ys* **by** *auto*
then have *x* # *xs* = (*x* # *zs*) @ *i* # *j* # *ys* **by** *simp*
then show *?thesis* **by** *blast*
qed
qed
qed

lemma *arcs-decomp-tail*:
arcs j l (*ys* @ [*i*]) = *arcs j i ys* @ [(*i*, *l*)]
by (*induction ys arbitrary*: *j*) *auto*

lemma *arcs-decomp*: *xs* = *ys* @ *y* # *zs* ⟹ *arcs x z xs* = *arcs x y ys* @ *arcs y z zs*
by (*induction ys arbitrary*: *x xs*) *simp+*

lemma *distinct-arcs-ex*:
distinct xs ⟹ *i* ∉ *set xs* ⟹ *xs* ≠ [] ⟹ ∃ *a b*. *a* ≠ *x* ∧ (*a*,*b*) ∈ *set* (*arcs i j xs*)
apply (*induction xs arbitrary*: *i*)
apply *simp*
subgoal for *a xs i*
apply (*cases xs*)
apply (*simp*, *metis*)
by *auto*
done

lemma *cycle-rotate-2-aux*:
(*i*, *j*) ∈ *set* (*arcs a b* (*xs* @ [*c*])) ⟹ (*i*,*j*) ≠ (*c*,*b*) ⟹ (*i*, *j*) ∈ *set* (*arcs a c xs*)
by (*induction xs arbitrary*: *a*) *auto*

lemma *arcs-set-elem1*:
assumes *j* ≠ *k k* ∈ *set* (*i* # *xs*)
shows ∃ *l*. (*k*, *l*) ∈ *set* (*arcs i j xs*) **using** *assms*
by (*induction xs arbitrary*: *i*) *auto*

lemma *arcs-set-elem2*:
assumes *i* ≠ *k k* ∈ *set* (*j* # *xs*)
shows ∃ *l*. (*l*, *k*) ∈ *set* (*arcs i j xs*) **using** *assms*
proof (*induction xs arbitrary*: *i*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs*)
then show *?case* **by** (*cases k* = *x*) *auto*

qed

2.2 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 (*induction xs arbitrary*: *i j*)
  case *Nil* **thus** *?case* **by** *auto*
next
  case (*Cons x xs*)
  have *M i x* + *len M x j xs* + (*M i x* + *len M x j xs*) = *M i x* + (*len M x j xs* + *M i x*) + *len M x j xs*
    by (*simp add*: *add.assoc*)
  also have ... = *M i x* + (*M i x* + *len M x j xs*) + *len M x j xs* **by** (*simp add*: *comm*)
  also have ... = (*M i x* + *M i x*) + (*len M x j xs* + *len M x j xs*) **by** (*simp add*: *add.assoc*)
  finally have *M i x* + *len M x j xs* + (*M i x* + *len M x j xs*)
          = (*M i x* + *M i x*) + *len* (λ*i j*. *M i j* + *M i j*) *x j xs*
  using *Cons* **by** *simp*
  thus *?case* **by** *simp*
qed

2.3 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* **using** *assms*
proof −
  assume *A*: *length xs* > *1* (*i*, *j*) ∈ *arcs' xs*
  from *arcs'-decomp*[*OF this*] **obtain** *ys zs* **where** *xs*: *xs* = *zs* @ *i* # *j* # *ys* **by** *blast*
  from *len-decomp*[*OF this*, *of M a a*]
  have *len M a a xs* = *len M a i zs* + *len M i a* (*j* # *ys*) **.**
  also have ... = *len M i a* (*j* # *ys*) + *len M a i zs* **by** (*simp add*: *comm*)
  also from *len-comp*[*of M i i j* # *ys a zs*] **have** ... = *len M i i* (*j* # *ys* @ *a* # *zs*) **by** *auto*
  finally show *?thesis* **using** *xs* **by** *auto*
qed

```
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
using assms proof −
  assume A:xs ≠ [] (i, j) ∈ set (arcs a a xs)
  { fix ys assume A:a = i xs = j # ys
    then have ?thesis by auto
  } note * = this
  { fix b ys assume A: a = j xs = ys @ [i]
    have len M j j (ys @ [i]) = M i j + len M j i ys
      using len-decomp[of ys @ [i] ys i [] M j j] by (auto simp: comm)
    with A have ?thesis
      by auto
  } note ** = this
  { assume length xs = 1
    then obtain b where xs: xs = [b] by (metis One-nat-def length-0-conv
length-Suc-conv)
    with A(2) have a = i ∧ b = j ∨ a = j ∧ b = i by auto
    then have ?thesis using * ** xs by auto
  } note *** = this
  show ?thesis
  proof (cases length xs = 0)
    case True with A show ?thesis by auto
  next
    case False
    thus ?thesis
    proof (cases length xs = 1, goal-cases)
      case True with *** show ?thesis by auto
    next
      case 2
      hence length xs > 1 by linarith
      then obtain b c ys where ys:xs = b # ys @ [c]
    by (metis One-nat-def assms(1) 2(2) length-0-conv length-Cons list.exhaust
rev-exhaust)
      thus ?thesis
      proof (cases (i,j) = (a,b), goal-cases)
        case True
        with ys * show ?thesis by auto
      next
        case False
        then show ?thesis
```

```
        proof (cases (i,j) = (c,a), goal-cases)
          case True
          with ys ** show ?thesis by auto
        next
          case 2
          with A(2) ys have (i, j) ∈ arcs′ xs
          using cycle-rotate-2-aux by (auto simp: arcs′-def)
           from cycle-rotate[OF ‹length xs > 1› this, of M a] show ?thesis
by auto
        qed
      qed
    qed
  qed
qed

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
using assms
proof −
  assume A: length xs > 1 (i, j) ∈ arcs′ xs
  from arcs′-decomp[OF this] obtain ys zs where xs: xs = zs @ i # j #
ys by blast
  from len-decomp[OF this, of M a a]
  have len M a a xs = len M a i zs + len M i a (j # ys) .
  also have . . . = len M i a (j # ys) + len M a i zs by (simp add: comm)
  also from len-comp[of M i i j # ys a zs] have . . . = len M i i (j # ys @
a # zs) by auto
  finally show ?thesis
  using xs arcs-decomp[OF xs, of a a] arcs-decomp[of j # ys @ a # zs j #
ys a zs i i] by force
qed

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 −
```

note *A* = *assms*
{ fix *ys* **assume** *A*:*a* = *i xs* = *j* # *ys*
then have *?thesis* **by** *auto*
} note ∗ = *this*
{ fix *b ys* **assume** *A*:*a* = *j xs* = *ys* @ [*i*]
have *len M j j* (*ys* @ [*i*]) = *M i j* + *len M j i ys*
using *len-decomp*[*of ys* @ [*i*] *ys i* [] *M j j*] **by** (*auto simp*: *comm*)
moreover have *arcs j j* (*ys* @ [*i*]) = *arcs j i ys* @ [(*i*, *j*)] **using** *arcs-decomp-tail* **by** *auto*
ultimately have *?thesis* **using** *A* **by** *auto*
} note ∗∗ = *this*
{ assume *length xs* = *1*
then obtain *b* **where** *xs*: *xs* = [*b*] **by** (*metis One-nat-def length-0-conv length-Suc-conv*)
with *A*(*2*) **have** *a* = *i* ∧ *b* = *j* ∨ *a* = *j* ∧ *b* = *i* **by** *auto*
then have *?thesis* **using** ∗ ∗∗ *xs* **by** *auto*
} note ∗∗∗ = *this*
show *?thesis*
proof (*cases length xs* = *0*)
case *True* **with** *A* **show** *?thesis* **by** *auto*
next
case *False*
thus *?thesis*
proof (*cases length xs* = *1*, *goal-cases*)
case *True* **with** ∗∗∗ **show** *?thesis* **by** *auto*
next
case *2*
hence *length xs* > *1* **by** *linarith*
then obtain *b c ys* **where** *ys*:*xs* = *b* # *ys* @ [*c*]
by (*metis One-nat-def assms*(*1*) *2*(*2*) *length-0-conv length-Cons list.exhaust rev-exhaust*)
thus *?thesis*
proof (*cases* (*i*,*j*) = (*a*,*b*))
case *True*
with *ys* ∗ **show** *?thesis* **by** *blast*
next
case *False*
then show *?thesis*
proof (*cases* (*i*,*j*) = (*c*,*a*), *goal-cases*)
case *True*
with *ys* ∗∗ **show** *?thesis* **by** *force*
next
case *2*
with *A*(*2*) *ys* **have** (*i*, *j*) ∈ *arcs′ xs*

using *cycle-rotate-2-aux* **by** (*auto simp add*: *arcs'-def*)
from *cycle-rotate-len-arcs*[*OF* ‹*length xs* > *1*› *this*, *of M a*] **show** *?thesis* **by** *auto*
qed
qed
qed
qed
qed

2.4 More on Cycle-Freeness

lemma *cyc-free-diag-dest*:
assumes *cyc-free M n i* ≤ *n set xs* ⊆ {*0..n*}
shows *len M i i xs* ≥ *0*
using *assms* **by** *auto*

lemma *cycle-free-0-0*:
fixes *M* :: (*'a*::*linordered-ab-monoid-add*) *mat*
assumes *cycle-free M n*
shows *M 0 0* ≥ *0*
using *cyc-free-diag-dest*[*OF cycle-free-diag-dest*[*OF assms*], *of 0* []] **by** *auto*

2.5 Helper Lemmas for Bouyer's Theorem on Approximation

lemma *aux1*: *i* ≤ *n* ⟹ *j* ≤ *n* ⟹ *set xs* ⊆ {*0..n*} ⟹ (*a*,*b*) ∈ *set* (*arcs i j xs*) ⟹ *a* ≤ *n* ∧ *b* ≤ *n*
by (*induction xs arbitrary*: *i*) *auto*

lemma *arcs-distinct1*:
i ∉ *set xs* ⟹ *j* ∉ *set xs* ⟹ *distinct xs* ⟹ *xs* ≠ [] ⟹ (*a*,*b*) ∈ *set* (*arcs i j xs*) ⟹ *a* ≠ *b*
apply (*induction xs arbitrary*: *i*)
apply *fastforce*
subgoal for *a' xs i*
by (*cases xs*) *auto*
done

lemma *arcs-distinct2*:
i ∉ *set xs* ⟹ *j* ∉ *set xs* ⟹ *distinct xs* ⟹ *i* ≠ *j* ⟹ (*a*,*b*) ∈ *set* (*arcs i j xs*) ⟹ *a* ≠ *b*
by (*induction xs arbitrary*: *i*) *auto*

lemma *arcs-distinct3*: *distinct* (*a* # *b* # *c* # *xs*) ⟹ (*i*,*j*) ∈ *set* (*arcs a b*

xs) ⟹ *i* ≠ *c* ∧ *j* ≠ *c*
by (*induction xs arbitrary*: *a*) *force*+

lemma *arcs-elem*:
  assumes (*a*, *b*) ∈ *set* (*arcs i j xs*) **shows** *a* ∈ *set* (*i* # *xs*) *b* ∈ *set* (*j* # *xs*)
using *assms* **by** (*induction xs arbitrary*: *i*) *auto*

lemma *arcs-distinct-dest1*:
  distinct (*i* # *a* # *zs*) ⟹ (*b*,*c*) ∈ *set* (*arcs a j zs*) ⟹ *b* ≠ *i*
using *arcs-elem* **by** *fastforce*

lemma *arcs-distinct-fix*:
  distinct (*a* # *x* # *xs* @ [*b*]) ⟹ (*a*,*c*) ∈ *set* (*arcs a b* (*x* # *xs*)) ⟹ *c* = *x*
using *arcs-elem*(*1*) **by** *fastforce*

lemma *disjE3*: *A* ∨ *B* ∨ *C* ⟹ (*A* ⟹ *G*) ⟹ (*B* ⟹ *G*) ⟹ (*C* ⟹ *G*) ⟹ *G*
by *auto*

lemma *arcs-predecessor*:
  assumes (*a*, *b*) ∈ *set* (*arcs i j xs*) *a* ≠ *i*
  shows ∃ *c*. (*c*, *a*) ∈ *set* (*arcs i j xs*) **using** *assms*
by (*induction xs arbitrary*: *i*) *auto*

lemma *arcs-successor*:
  assumes (*a*, *b*) ∈ *set* (*arcs i j xs*) *b* ≠ *j*
  shows ∃ *c*. (*b*,*c*) ∈ *set* (*arcs i j xs*) **using** *assms*
  apply (*induction xs arbitrary*: *i*)
   apply *simp*
  subgoal for *aa xs i*
    by (*cases xs*) *auto*
  done

lemma *arcs-predecessor′*:
  assumes (*a*, *b*) ∈ *set* (*arcs i j* (*x* # *xs*)) (*a*,*b*) ≠ (*i*, *x*)
  shows ∃ *c*. (*c*, *a*) ∈ *set* (*arcs i j* (*x* # *xs*)) **using** *assms*
by (*induction xs arbitrary*: *i x*) *auto*

lemma *arcs-cases*:
  assumes (*a*, *b*) ∈ *set* (*arcs i j xs*) *xs* ≠ []
  shows (∃ *ys*. *xs* = *b* # *ys* ∧ *a* = *i*) ∨ (∃ *ys*. *xs* = *ys* @ [*a*] ∧ *b* = *j*)
    ∨ (∃ *c d ys*. (*a*,*b*) ∈ *set* (*arcs c d ys*) ∧ *xs* = *c* # *ys* @ [*d*])
using *assms*

```
proof (induction xs arbitrary: i)
  case Nil then show ?case by auto
next
  case (Cons x xs)
  show ?case
  proof (cases (a, b) = (i, x))
    case True
    with Cons.prems show ?thesis by auto
  next
    case False
    note F = this
    show ?thesis
    proof (cases xs = [])
      case True
      with F Cons.prems show ?thesis by auto
    next
      case False
      from F Cons.prems have (a, b) ∈ set (arcs x j xs) by auto
      from Cons.IH[OF this False] have
        (∃ ys. xs = b # ys ∧ a = x) ∨ (∃ ys. xs = ys @ [a] ∧ b = j)
         ∨ (∃ c d ys. (a, b) ∈ set (arcs c d ys) ∧ xs = c # ys @ [d])
      .
      then show ?thesis
      proof (rule disjE3, goal-cases)
        case 1
        from 1 obtain ys where *: xs = b # ys ∧ a = x by auto
        show ?thesis
        proof (cases ys = [])
          case True
          with * show ?thesis by auto
        next
          case False
              then obtain z zs where zs: ys = zs @ [z] by (metis ap-
pend-butlast-last-id)
          with * show ?thesis by auto
        qed
      next
        case 2 then show ?case by auto
      next
        case 3 with False show ?case by auto
      qed
    qed
  qed
qed
```

```
lemma arcs-cases':
  assumes (a, b) ∈ set (arcs i j xs) xs ≠ []
  shows (∃ ys. xs = b # ys ∧ a = i) ∨ (∃ ys. xs = ys @ [a] ∧ b = j) ∨
(∃ ys zs. xs = ys @ a # b # zs)
using assms
proof (induction xs arbitrary: i)
  case Nil then show ?case by auto
next
  case (Cons x xs)
  show ?case
  proof (cases (a, b) = (i, x))
    case True
    with Cons.prems show ?thesis by auto
  next
    case False
    note F = this
    show ?thesis
    proof (cases xs = [])
      case True
      with F Cons.prems show ?thesis by auto
    next
      case False
      from F Cons.prems have (a, b) ∈ set (arcs x j xs) by auto
      from Cons.IH[OF this False] have
        (∃ ys. xs = b # ys ∧ a = x) ∨ (∃ ys. xs = ys @ [a] ∧ b = j)
         ∨ (∃ ys zs. xs = ys @ a # b # zs)
      .
      then show ?thesis
      proof (rule disjE3, goal-cases)
        case 1
        from 1 obtain ys where *: xs = b # ys ∧ a = x by auto
        show ?thesis
        proof (cases ys = [])
          case True
          with * show ?thesis by auto
        next
          case False
              then obtain z zs where zs: ys = zs @ [z] by (metis ap-
pend-butlast-last-id)
          with * show ?thesis by auto
        qed
      next
        case 2 then show ?case by auto
```

```
      next
        case 3
        then obtain ys zs where xs = ys @ a # b # zs by auto
        then have x # xs = (x # ys) @ a # b # zs by auto
        then show ?thesis by blast
      qed
    qed
  qed
qed

lemma arcs-successor':
  assumes (a, b) ∈ set (arcs i j xs) b ≠ j
  shows ∃ c. xs = [b] ∧ a = i ∨ (∃ ys. xs = b # c # ys ∧ a = i) ∨ (∃ ys.
xs = ys @ [a,b] ∧ c = j)
      ∨ (∃ ys zs. xs = ys @ a # b # c # zs)
using assms
proof (induction xs arbitrary: i)
  case Nil then show ?case by auto
next
  case (Cons x xs)
  show ?case
  proof (cases (a, b) = (i, x))
    case True
    with Cons.prems show ?thesis by (cases xs) auto
  next
    case False
    note F = this
    show ?thesis
    proof (cases xs = [])
      case True
      with F Cons.prems show ?thesis by auto
    next
      case False
      from F Cons.prems have (a, b) ∈ set (arcs x j xs) by auto
      from Cons.IH[OF this ‹b ≠ j›] obtain c where c:
        xs = [b] ∧ a = x ∨ (∃ ys. xs = b # c # ys ∧ a = x) ∨ (∃ ys. xs =
ys @ [a, b] ∧ c = j)
        ∨ (∃ ys zs. xs = ys @ a # b # c # zs)
      ..
      then show ?thesis
      proof (standard, goal-cases)
        case 1 with Cons.prems show ?case by auto
      next
        case 2
```

```
        then show ?thesis
        proof (rule disjE3, goal-cases)
          case 1
          from 1 obtain ys where *: xs = b # ys ∧ a = x by auto
          show ?thesis
          proof (cases ys = [])
            case True
            with * show ?thesis by auto
          next
            case False
            then obtain z zs where zs: ys = z # zs by (cases ys) auto
            with * show ?thesis by fastforce
          qed
        next
          case 2 then show ?case by auto
        next
          case 3
          then obtain ys zs where xs = ys @ a # b # c # zs by auto
          then have x # xs = (x # ys) @ a # b # c # zs by auto
          then show ?thesis by blast
        qed
      qed
    qed
  qed
qed

lemma list-last:
  xs = [] ∨ (∃ y ys. xs = ys @ [y])
by (induction xs) auto

lemma arcs-predecessor'':
  assumes (a, b) ∈ set (arcs i j xs) a ≠ i
 shows ∃ c. xs = [a] ∨ (∃ ys. xs = a # b # ys) ∨ (∃ ys. xs = ys @ [c,a]
∧ b = j)
      ∨ (∃ ys zs. xs = ys @ c # a # b # zs)
using assms
proof (induction xs arbitrary: i)
  case Nil then show ?case by auto
next
  case (Cons x xs)
  show ?case
  proof (cases (a, b) = (i, x))
    case True
    with Cons.prems show ?thesis by (cases xs) auto
```

```
 next
   case False
   note F = this
   show ?thesis
   proof (cases xs = [])
     case True
     with F Cons.prems show ?thesis by auto
   next
     case False
     from F Cons.prems have *: (a, b) ∈ set (arcs x j xs) by auto
     from False obtain y ys where xs: xs = y # ys by (cases xs) auto
     show ?thesis
     proof (cases (a,b) = (x,y))
       case True with * xs show ?thesis by auto
     next
       case False
       with * xs have **: (a, b) ∈ set (arcs y j ys) by auto
       show ?thesis
       proof (cases ys = [])
         case True with ** xs show ?thesis by force
       next
         case False
         from arcs-cases'[OF ** this] obtain ws zs where ***:
           ys = b # ws ∧ a = y ∨ ys = ws @ [a] ∧ b = j ∨ ys = ws @ a #
b # zs
         by auto
         then show ?thesis
         proof (elim disjE, goal-cases)
           case 1
           then show ?case using xs by blast
         next
           case 2
           then have ∃ y ys. ws = ys @ [y] if ws ≠ []
             using list-last[of ws] that by fastforce
           with 2 show ?case
             using xs by (cases ws = []) auto
         next
           case 3
           then have x # xs = [x] @ y # a # b # zs if ws = []
             using that by (simp add: xs)
           with 3 show ?case
             apply (cases ws = [])
               apply blast
            by (metis append.left-neutral append-Cons append-assoc list-last
```

xs)
          qed
        qed
      qed
    qed
  qed
qed

lemma *arcs-ex-middle*:
  ∃ *b*. (*a*, *b*) ∈ *set* (*arcs i j* (*ys* @ *a* # *xs*))
by (*induction xs arbitrary*: *i ys*) (*auto simp*: *arcs-decomp*)

lemma *arcs-ex-head*:
  ∃ *b*. (*i*, *b*) ∈ *set* (*arcs i j xs*)
by (*cases xs*) *auto*

2.5.1 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*] **by** *simp*
lemma *successive* (λ *x*. *x* > (*0* :: *nat*)) [*Suc 0*] **by** *simp*
lemma *successive* (λ *x*. *x* > (*0* :: *nat*)) [*Suc 0*, *0*, *Suc 0*] **by** *simp*
lemma ¬ *successive* (λ *x*. *x* > (*0* :: *nat*)) [*Suc 0*, *0*, *Suc 0*, *Suc 0*] **by** *simp*
lemma ¬ *successive* (λ *x*. *x* > (*0* :: *nat*)) [*Suc 0*, *0*, *0*, *Suc 0*, *Suc 0*] **by** *simp*
lemma *successive* (λ *x*. *x* > (*0* :: *nat*)) [*Suc 0*, *0*, *Suc 0*, *0*, *Suc 0*] **by** *simp*
lemma ¬ *successive* (λ *x*. *x* > (*0* :: *nat*)) [*Suc 0*, *Suc 0*, *0*, *Suc 0*] **by** *simp*
lemma *successive* (λ *x*. *x* > (*0* :: *nat*)) [*0*, *0*, *Suc 0*, *0*] **by** *simp*

lemma *successive-step*: *successive P* (*x* # *xs*) ⟹ ¬ *P x* ⟹ *successive P xs*
  apply (*cases xs*)
   apply *simp*
  subgoal for *y ys*
    by (*cases ys*) *auto*
  done

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

apply (*cases xs*)
 apply *simp*
subgoal for *z zs*
 by (*cases zs*) *auto*
done

lemma *successive-stepI*:
successive P xs ⟹ ¬ *P x* ⟹ *successive P* (*x* # *xs*)
by (*cases xs*) *auto*

lemmas *list-two-induct*[*case-names Nil Single Cons*] = *induct-list012*

lemma *successive-end-1*:
successive P xs ⟹ ¬ *P x* ⟹ *successive P* (*xs* @ [*x*])
by (*induction* - *xs rule*: *list-two-induct*) *auto*

lemma *successive-ends-1*:
successive P xs ⟹ ¬ *P x* ⟹ *successive P ys* ⟹ *successive P* (*xs* @ *x* # *ys*)
by (*induction* - *xs rule*: *list-two-induct*) (*fastforce intro*: *successive-stepI*)+

lemma *successive-ends-1'*:
successive P xs ⟹ ¬ *P x* ⟹ *P y* ⟹ ¬ *P z* ⟹ *successive P ys* ⟹ *successive P* (*xs* @ *x* # *y* # *z* # *ys*)
by (*induction* - *xs rule*: *list-two-induct*) (*fastforce intro*: *successive-stepI*)+

lemma *successive-end-2*:
successive P xs ⟹ ¬ *P x* ⟹ *successive P* (*xs* @ [*x*,*y*])
by (*induction* - *xs rule*: *list-two-induct*) *auto*

lemma *successive-end-2'*:
successive P (*xs* @ [*x*]) ⟹ ¬ *P x* ⟹ *successive P* (*xs* @ [*x*,*y*])
by (*induction* - *xs rule*: *list-two-induct*) *auto*

lemma *successive-end-3*:
successive P (*xs* @ [*x*]) ⟹ ¬ *P x* ⟹ *P y* ⟹ ¬ *P z* ⟹ *successive P* (*xs* @ [*x*,*y*,*z*])
by (*induction* - *xs rule*: *list-two-induct*) *auto*

lemma *successive-step-rev*:
successive P (*xs* @ [*x*]) ⟹ ¬ *P x* ⟹ *successive P xs*
by (*induction* - *xs rule*: *list-two-induct*) *auto*

lemma *successive-glue*:

successive P (*zs* @ [*z*]) ⟹ *successive P* (*x* # *xs*) ⟹ ¬ *P z* ∨ ¬ *P x* ⟹ *successive P* (*zs* @ [*z*] @ *x* # *xs*)
proof *goal-cases*
  case *A*: *1*
  show *?thesis*
  proof (*cases P x*)
    case *False*
    with *A*(*1*,*2*) *successive-ends-1 successive-step* **show** *?thesis* **by** *fastforce*
  next
    case *True*
    with *A*(*1*,*3*) *successive-step-rev* **have** ¬ *P z successive P zs* **by** *fastforce+*
    with *A*(*2*) *successive-ends-1* **show** *?thesis* **by** *fastforce*
  qed
qed

lemma *successive-glue′*:
  successive P (*zs* @ [*y*]) ∧ ¬ *P z* ∨ *successive P zs* ∧ ¬ *P y*
  ⟹ *successive P* (*x* # *xs*) ∧ ¬ *P w* ∨ *successive P xs* ∧ ¬ *P x*
  ⟹ ¬ *P z* ∨ ¬ *P w* ⟹ *successive P* (*zs* @ *y* # *z* # *w* # *x* # *xs*)
by (*metis append-Cons append-Nil successive.simps*(*3*) *successive-ends-1 successive-glue successive-stepI*)

lemma *successive-dest-head*:
  xs = *w* # *x* # *ys* ⟹ *successive P xs* ⟹ *successive P* (*x* # *xs*) ∧ ¬ *P w* ∨ *successive P xs* ∧ ¬ *P x*
by *auto*

lemma *successive-dest-tail*:
  xs = *zs* @ [*y*,*z*] ⟹ *successive P xs*
  ⟹ *successive P* (*zs* @ [*y*]) ∧ ¬ *P z* ∨ *successive P zs* ∧ ¬ *P y*
  apply (*induction - xs arbitrary*: *zs rule*: *list-two-induct*)
    apply *simp+*
  subgoal for - - - *zs*
    apply (*cases zs*)
     apply *simp*
    subgoal for - *ws*
      by (*cases ws*) *auto*
    done
  done

lemma *successive-split*:
  xs = *ys* @ *zs* ⟹ *successive P xs* ⟹ *successive P ys* ∧ *successive P zs*
  apply (*induction - xs arbitrary*: *ys rule*: *list-two-induct*)
    apply *simp*

subgoal for - *ys*
by (*cases ys*; *simp*)
subgoal for - - - *ys*
apply (*cases ys*; *simp*)
subgoal for *list*
by (*cases list*) (*auto intro*: *successive-stepI*)
done
done

lemma *successive-decomp*:
$xs = x \# ys @ zs @ [z] \Longrightarrow successive\ P\ xs \Longrightarrow \neg\ P\ x \vee \neg\ P\ z \Longrightarrow successive\ P\ (zs @ [z] @ (x \# ys))$
by (*metis Cons-eq-appendI successive-glue successive-split*)

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 $\exists\ c.\ (xs = [a] \wedge c = i \vee (\exists\ ys.\ xs = a \# b \# ys \wedge c = i) \vee (\exists\ ys.\ xs = ys @ [c,a] \wedge b = j)$
$\vee (\exists\ ys\ zs.\ xs = ys @ c \# a \# b \# zs)) \wedge \neg\ P\ (c,a)$
proof −
from *arcs-predecessor''*[*OF assms*(*1*,*2*)] **obtain** *c* **where** *c*:
$xs = [a] \vee (\exists ys.\ xs = a \# b \# ys) \vee (\exists ys.\ xs = ys @ [c, a] \wedge b = j)$
$\vee (\exists ys\ zs.\ xs = ys @ c \# a \# b \# zs)$
by *auto*
then show *?thesis*
proof (*safe*, *goal-cases*)
case *1*
with *assms* **have** $arcs\ i\ j\ xs = [(i, a), (a, j)]$ **by** *auto*
with *assms* **have** $\neg\ P\ (i, a)$ **by** *auto*
with *1* **show** *?case* **by** *simp*
next
case *2*
with *assms* **have** $\neg\ P\ (i, a)$ **by** *fastforce*
with *2* **show** *?case* **by** *auto*
next
case *3*
with *assms* **have** $\neg\ P\ (c, a)$ **using** *arcs-decomp successive-dest-tail* **by** *fastforce*
with *3* **show** *?case* **by** *auto*
next
case *4*
with *assms*(*3*,*4*) **have** $\neg\ P\ (c, a)$ **using** *arcs-decomp successive-split* **by** *fastforce*

with *4* **show** *?case* **by** *auto*
qed
qed

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 $\exists\ c.\ (xs = [b] \wedge c = j \vee (\exists\ ys.\ xs = b \# c \# ys) \vee (\exists\ ys.\ xs = ys$
$@ [a,b] \wedge c = j)$
$\vee (\exists\ ys\ zs.\ xs = ys @ a \# b \# c \# zs)) \wedge \neg\ P\ (b,c)$
proof −
from *arcs-successor'*[*OF assms*(*1*,*2*)] **obtain** *c* **where** *c*:
$xs = [b] \wedge a = i \vee (\exists ys.\ xs = b \# c \# ys \wedge a = i) \vee (\exists ys.\ xs = ys\ @$
$[a,\ b] \wedge c = j)$
$\vee (\exists ys\ zs.\ xs = ys @ a \# b \# c \# zs)$
..
then show *?thesis*
proof (*safe*, *goal-cases*)
case *1*
with *assms*(*1*,*2*) **have** *arcs i j xs* $= [(a,b), (b,j)]$ **by** *auto*
with *assms* **have** $\neg\ P\ (b,j)$ **by** *auto*
with *1* **show** *?case* **by** *simp*
next
case *2*
with *assms* **have** $\neg\ P\ (b,\ c)$ **by** *fastforce*
with *2* **show** *?case* **by** *auto*
next
case *3*
with *assms* **have** $\neg\ P\ (b,\ c)$ **using** *arcs-decomp successive-dest-tail* **by**
fastforce
with *3* **show** *?case* **by** *auto*
next
case *4*
with *assms*(*3*,*4*) **have** $\neg\ P\ (b,\ c)$ **using** *arcs-decomp successive-split* **by**
fastforce
with *4* **show** *?case* **by** *auto*
qed
qed

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 ≡ λ i j. min (A i j) (B i j)
  assumes canonical A n
  assumes set xs ⊆ {0..n}
  assumes i ≤ n j ≤ n
  shows ∃ ys. len M i j ys ≤ len M i j xs ∧ set ys ⊆ {0..n}
             ∧ successive (λ (a, b). M a b = A a b) (arcs i j ys)
using assms
proof (induction xs arbitrary: i rule: list-two-induct)
  case Nil show ?case by fastforce
next
  case 2: (Single x i)
  show ?case
  proof (cases M i x = A i x ∧ M x j = A x j)
    case False
    then have successive (λ(a, b). M a b = A a b) (arcs i j [x]) by auto
    with 2 show ?thesis by blast
  next
    case True
    with 2 have M i j ≤ M i x + M x j unfolding min-def by fastforce
    with 2(3−) show ?thesis apply simp apply (rule exI[where x = []])
by auto
  qed
next
  case 3: (Cons x y xs i)
  show ?case
  proof (cases M i y ≤ M i x + M x y, goal-cases)
    case 1
    from 3 obtain ys where
      len M i j ys ≤ len M i j (y # xs) ∧ set ys ⊆ {0..n}
       ∧ successive (λa. case a of (a, b) ⇒ M a b = A a b) (arcs i j ys)
    by fastforce
    moreover from 1 have len M i j (y # xs) ≤ len M i j (x # y # xs)
    using add-mono by (auto simp: add.assoc[symmetric])
    ultimately show ?case by force
  next
    case False
    { assume M i x = A i x M x y = A x y
      with 3(3−) have A i y ≤ M i x + M x y by auto
     then have M i y ≤ M i x + M x y unfolding M-def min-def by auto
    } note ∗ = this
    with False have M i x ≠ A i x ∨ M x y ≠ A x y by auto
    then show ?thesis
    proof (standard, goal-cases)
```

```
    case 1
    from 3 obtain ys where ys:
      len M x j ys ≤ len M x j (y # xs) set ys ⊆ {0..n}
      successive (λa. case a of (a, b) ⇒ M a b = A a b) (arcs x j ys)
    by force+
    from 1 successive-stepI[OF ys(3), of (i, x)] have
     successive (λa. case a of (a, b) ⇒ M a b = A a b) (arcs i j (x # ys))
    by auto
    moreover have len M i j (x # ys) ≤ len M i j (x # y # xs) using
add-mono-right[OF ys(1)]
    by auto
    ultimately show ?case using 3(5) ys(2) by fastforce
  next
    case 2
    from 3 obtain ys where ys:
      len M y j ys ≤ len M y j xs set ys ⊆ {0..n}
      successive (λa. case a of (a, b) ⇒ M a b = A a b) (arcs y j ys)
    by force+
    from this(3) 2 have
      successive (λa. case a of (a, b) ⇒ M a b = A a b) (arcs i j (x # y
# ys))
    by simp
    moreover from add-mono-right[OF ys(1)] have
      len M i j (x # y # ys) ≤ len M i j (x # y # xs)
    by (auto simp: add.assoc[symmetric])
    ultimately show ?thesis using ys(2) 3(5) by fastforce
  qed
 qed
qed

lemma canonical-successive-distinct:
 fixes A B
 defines M ≡ λ i j. min (A i j) (B i j)
 assumes canonical A n
 assumes set xs ⊆ {0..n}
 assumes i ≤ n j ≤ n
 assumes distinct xs i ∉ set xs j ∉ set xs
 shows ∃ ys. len M i j ys ≤ len M i j xs ∧ set ys ⊆ set xs
           ∧ successive (λ (a, b). M a b = A a b) (arcs i j ys)
           ∧ distinct ys ∧ i ∉ set ys ∧ j ∉ set ys
using assms
proof (induction xs arbitrary: i rule: list-two-induct)
 case Nil show ?case by fastforce
next
```

```
  case 2: (Single x i)
  show ?case
  proof (cases M i x = A i x ∧ M x j = A x j)
    case False
    then have successive (λ(a, b). M a b = A a b) (arcs i j [x]) by auto
    with 2 show ?thesis by blast
  next
    case True
    with 2 have M i j ≤ M i x + M x j unfolding min-def by fastforce
    with 2(3−) show ?thesis apply simp apply (rule exI[where x = []])
by auto
  qed
next
  case 3: (Cons x y xs i)
  show ?case
  proof (cases M i y ≤ M i x + M x y)
    case 1: True
    from 3(2)[OF 3(3,4)] 3(5−10) obtain ys where ys:
      len M i j ys ≤ len M i j (y # xs) set ys ⊆ set (x # y # xs)
       successive (λa. case a of (a, b) ⇒ M a b = A a b) (arcs i j ys)
       distinct ys ∧ i ∉ set ys ∧ j ∉ set ys
    by fastforce
    moreover from 1 have len M i j (y # xs) ≤ len M i j (x # y # xs)
    using add-mono by (auto simp: add.assoc[symmetric])
    ultimately have len M i j ys ≤ len M i j (x # y # xs) by auto
    then show ?thesis using ys(2−) by blast
  next
    case False
    { assume M i x = A i x M x y = A x y
      with 3(3−) have A i y ≤ M i x + M x y by auto
     then have M i y ≤ M i x + M x y unfolding M-def min-def by auto
    } note ∗ = this
    with False have M i x ≠ A i x ∨ M x y ≠ A x y by auto
    then show ?thesis
    proof (standard, goal-cases)
      case 1
      from 3(2)[OF 3(3,4)] 3(5−10) obtain ys where ys:
        len M x j ys ≤ len M x j (y # xs) set ys ⊆ set (y # xs)
        successive (λa. case a of (a, b) ⇒ M a b = A a b) (arcs x j ys)
        distinct ys i ∉ set ys x ∉ set ys j ∉ set ys
      by fastforce
      from 1 successive-stepI[OF ys(3), of (i, x)] have
       successive (λa. case a of (a, b) ⇒ M a b = A a b) (arcs i j (x # ys))
      by auto
```

moreover have *len M i j* (*x* # *ys*) ≤ *len M i j* (*x* # *y* # *xs*) **using** *add-mono-right*[*OF ys*(*1*)]
by *auto*
moreover have *distinct* (*x* # *ys*) *i* ∉ *set* (*x* # *ys*) *j* ∉ *set* (*x* # *ys*) **using** *ys*(*4*−) *3*(*8*−)
by *auto*
moreover from *ys*(*2*) **have** *set* (*x* # *ys*) ⊆ *set* (*x* # *y* # *xs*) **by** *auto*
ultimately show *?case* **by** *fastforce*
next
case *2*
from *3*(*1*)[*OF 3*(*3*,*4*)] *3*(*5*−) **obtain** *ys* **where** *ys*:
len M y j ys ≤ *len M y j xs set ys* ⊆ *set xs*
successive (λ*a*. *case a of* (*a*, *b*) ⇒ *M a b* = *A a b*) (*arcs y j ys*)
distinct ys j ∉ *set ys y* ∉ *set ys i* ∉ *set ys x* ∉ *set ys*
by *fastforce*
from *this*(*3*) *2* **have**
successive (λ*a*. *case a of* (*a*, *b*) ⇒ *M a b* = *A a b*) (*arcs i j* (*x* # *y* # *ys*))
by *simp*
moreover from *add-mono-right*[*OF ys*(*1*)] **have**
len M i j (*x* # *y* # *ys*) ≤ *len M i j* (*x* # *y* # *xs*)
by (*auto simp*: *add.assoc*[*symmetric*])
moreover have *distinct* (*x* # *y* # *ys*) *i* ∉ *set* (*x* # *y* # *ys*) *j* ∉ *set* (*x* # *y* # *ys*)
using *ys*(*4*−) *3*(*8*−) **by** *auto*
ultimately show *?thesis* **using** *ys*(*2*) **by** *fastforce*
qed
qed
qed

lemma *successive-snd-last*: *successive P* (*xs* @ [*x*, *y*]) ⟹ *P y* ⟹ ¬ *P x*
by (*induction - xs rule*: *list-two-induct*) *auto*

lemma *canonical-shorten-rotate-neg-cycle*:
fixes *A B*
defines *M* ≡ λ *i j*. *min* (*A i j*) (*B i j*)
assumes *canonical A n*
assumes *set xs* ⊆ {*0*..*n*}
assumes *i* ≤ *n*
assumes *len M i i xs* < *0*
shows ∃ *j ys*. *len M j j ys* < *0* ∧ *set* (*j* # *ys*) ⊆ *set* (*i* # *xs*)
∧ *successive* (λ (*a*, *b*). *M a b* = *A a b*) (*arcs j j ys*)
∧ *distinct ys* ∧ *j* ∉ *set ys* ∧
(*ys* ≠ [] ⟶ *M j* (*hd ys*) ≠ *A j* (*hd ys*) ∨ *M* (*last ys*) *j* ≠ *A*

```
(last ys) j)
using assms
proof -
  note A = assms
  from negative-len-shortest[OF - A(5)] obtain j ys where ys:
    distinct (j # ys) len M j j ys < 0 j ∈ set (i # xs) set ys ⊆ set xs
  by blast
 from this(1,3) canonical-successive-distinct[OF A(2) subset-trans[OF this(4)
A(3)], of j j B] A(3,4)
  obtain zs where zs:
    len M j j zs ≤ len M j j ys
    set zs ⊆ set ys successive (λ(a, b). M a b = A a b) (arcs j j zs)
    distinct zs j ∉ set zs
  by (force simp: M-def)
  show ?thesis
  proof (cases zs = [])
    assume zs ≠ []
    then obtain w ws where ws: zs = w # ws by (cases zs) auto
    show ?thesis
    proof (cases ws = [])
      case False
      then obtain u us where us: ws = us @ [u] by (induction ws) auto
      show ?thesis
      proof (cases M j w = A j w ∧ M u j = A u j)
        case True
       have u ≤ n j ≤ n w ≤ n using us ws zs(2) ys(3,4) A(3,4) by auto
        with A(2) True have M u w ≤ M u j + M j w unfolding M-def
min-def by fastforce
        then have
          len M u u (w # us) ≤ len M j j zs
        using ws us by (simp add: len-comp comm) (auto intro: add-mono
simp: add.assoc[symmetric])
       moreover have set (u # w # us) ⊆ set (i # xs) using ws us zs(2)
ys(3,4) by auto
        moreover have distinct (w # us) u ∉ set (w # us) using ws us
zs(4) by auto
       moreover have successive (λ(a, b). M a b = A a b) (arcs u u (w #
us))
        proof (cases us)
          case Nil
          with zs(3) ws us True show ?thesis by auto
        next
          case (Cons v vs)
          with zs(3) ws us True have M w v ≠ A w v by auto
```

with *ws us Cons zs*(*3*) *True arcs-decomp-tail successive-split* **show** *?thesis* **by** (*simp*, *blast*)
qed
moreover have *M* (*last* (*w* # *us*)) *u* ≠ *A* (*last* (*w* # *us*)) *u*
proof (*cases us* = [])
case *T*: *True*
with *zs*(*3*) *ws us True* **show** *?thesis* **by** *auto*
next
case *False*
then obtain *v vs* **where** *vs*: *us* = *vs* @ [*v*] **by** (*induction us*) *auto*
with *ws us* **have** *arcs j j zs* = *arcs j v* (*w* # *vs*) @ [(*v*, *u*), (*u*,*j*)] **by** (*simp add*: *arcs-decomp*)
with *zs*(*3*) *True* **have** *M v u* ≠ *A v u*
using *successive-snd-last*[*of* λ(*a*, *b*). *M a b* = *A a b arcs j v* (*w* # *vs*)] **by** *auto*
with *vs* **show** *?thesis* **by** *simp*
qed
ultimately show *?thesis* **using** *zs*(*1*) *ys*(*2*)
by (*intro exI*[**where** *x* = *u*], *intro exI*[**where** *x* = *w* # *us*]) *fastforce*
next
case *False*
with *zs ws us ys* **show** *?thesis* **by** (*intro exI*[**where** *x* = *j*], *intro exI*[**where** *x* = *zs*]) *auto*
qed
next
case *True*
with *True ws zs ys* **show** *?thesis* **by** (*intro exI*[**where** *x* = *j*], *intro exI*[**where** *x* = *zs*]) *fastforce*
qed
next
case *True*
with *ys zs* **show** *?thesis* **by** (*intro exI*[**where** *x* = *j*], *intro exI*[**where** *x* = *zs*]) *fastforce*
qed
qed

lemma *successive-arcs-extend-last*:
successive P (*arcs i j xs*) ⟹ ¬ *P* (*i*, *hd xs*) ∨ ¬ *P* (*last xs*, *j*) ⟹ *xs* ≠ []
⟹ *successive P* (*arcs i j xs* @ [(*i*, *hd xs*)])
proof −
assume *a1*: ¬ *P* (*i*, *hd xs*) ∨ ¬ *P* (*last xs*, *j*)
assume *a2*: *successive P* (*arcs i j xs*)
assume *a3*: *xs* ≠ []

then have *f4*: ¬ *P* (*last xs*, *j*) ⟶ *successive P* (*arcs i* (*last xs*) (*butlast xs*))
using *a2* **by** (*metis* (*no-types*) *append-butlast-last-id arcs-decomp-tail successive-step-rev*)
have *f5*: *arcs i j xs* = *arcs i* (*last xs*) (*butlast xs*) @ [(*last xs*, *j*)]
using *a3* **by** (*metis* (*no-types*) *append-butlast-last-id arcs-decomp-tail*)
have ([] @ *arcs i j xs* @ [(*i*, *hd xs*)]) @ [(*i*, *hd xs*)] = *arcs i j xs* @ [(*i*, *hd xs*), (*i*, *hd xs*)]
by *simp*
then have *P* (*last xs*, *j*) ⟶ *successive P* (*arcs i j xs* @ [(*i*, *hd xs*)])
using *a2 a1* **by** (*metis* (*no-types*) *self-append-conv2 successive-end-2 successive-step-rev*)
then show *?thesis*
using *f5 f4 successive-end-2* **by** *fastforce*
qed

lemma *cycle-rotate-arcs*:
fixes *M* :: (*'a* :: *linordered-ab-monoid-add*) *mat*
assumes *length xs* > *1* (*i*, *j*) ∈ *arcs' xs*
shows ∃ *ys zs*. *set* (*arcs a a xs*) = *set* (*arcs i i* (*j* # *ys* @ *a* # *zs*)) ∧ *xs* = *zs* @ *i* # *j* # *ys* **using** *assms*
proof −
assume *A*: *length xs* > *1* (*i*, *j*) ∈ *arcs' xs*
from *arcs'-decomp*[*OF this*] **obtain** *ys zs* **where** *xs*: *xs* = *zs* @ *i* # *j* # *ys* **by** *blast*
with *arcs-decomp*[*OF this*, *of a a*] *arcs-decomp*[*of j* # *ys* @ *a* # *zs j* # *ys a zs i i*]
show *?thesis* **by** *force*
qed

lemma *cycle-rotate-len-arcs-successive*:
fixes *M* :: (*'a* :: *linordered-ab-monoid-add*) *mat*
assumes *length xs* > *1* (*i*, *j*) ∈ *arcs' xs successive P* (*arcs a a xs*) ¬ *P* (*a*, *hd xs*) ∨ ¬ *P* (*last xs*, *a*)
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*
∧ *successive P* (*arcs i i* (*j* # *ys* @ *a* # *zs*))
using *assms*
proof −
note *A* = *assms*
from *arcs'-decomp*[*OF A*(*1*,*2*)] **obtain** *ys zs* **where** *xs*: *xs* = *zs* @ *i* # *j* # *ys* **by** *blast*
note *arcs1* = *arcs-decomp*[*OF xs*, *of a a*]

note *arcs2* = *arcs-decomp*[*of j* # *ys* @ *a* # *zs j* # *ys a zs i i*]
have ∗:*successive P* (*arcs i i* (*j* # *ys* @ *a* # *zs*))
proof (*cases ys* = [])
case *True*
show *?thesis*
proof (*cases zs*)
case *Nil*
with *A*(*3*,*4*) *xs True* **show** *?thesis* **by** *auto*
next
case (*Cons z zs*′)
with *True arcs2 A*(*3*,*4*) *xs* **show** *?thesis* **apply** *simp*
by (*metis arcs.simps*(*1*,*2*) *arcs1 successive.simps*(*3*) *successive-split successive-step*)
qed
next
case *False*
then obtain *y ys*′ **where** *ys*: *ys* = *ys*′ @ [*y*] **by** (*metis append-butlast-last-id*)
show *?thesis*
proof (*cases zs*)
case *Nil*
with *A*(*3*,*4*) *xs ys* **have**
¬ *P* (*a*, *i*) ∨ ¬ *P* (*y*, *a*) *successive P* (*arcs a a* (*i* # *j* # *ys*′ @ [*y*]))
by *simp*+
from *successive-decomp*[*OF* - *this*(*2*,*1*)] **show** *?thesis* **using** *ys Nil arcs-decomp* **by** *fastforce*
next
case (*Cons z zs*′)
with *A*(*3*,*4*) *xs ys* **have**
¬ *P* (*a*, *z*) ∨ ¬ *P* (*y*, *a*) *successive P* (*arcs a a* (*z* # *zs*′ @ *i* # *j* # *ys*′ @ [*y*]))
by *simp*+
from *successive-decomp*[*OF* - *this*(*2*,*1*)] **show** *?thesis* **using** *ys Cons arcs-decomp* **by** *fastforce*
qed
qed
from *len-decomp*[*OF xs*, *of M a a*] **have** *len M a a xs* = *len M a i zs* + *len M i a* (*j* # *ys*) **.**
also have . . . = *len M i a* (*j* # *ys*) + *len M a i zs* **by** (*simp add*: *comm*)
also from *len-comp*[*of M i i j* # *ys a zs*] **have** . . . = *len M i i* (*j* # *ys* @ *a* # *zs*) **by** *auto*
finally show *?thesis*
using ∗ *xs arcs-decomp*[*OF xs*, *of a a*] *arcs-decomp*[*of j* # *ys* @ *a* # *zs j* # *ys a zs i i*] **by** *force*
qed

lemma *successive-successors*:
  xs = *ys* @ *a* # *b* # *c* # *zs* ⟹ *successive P* (*arcs i j xs*) ⟹ ¬ *P* (*a*,*b*) ∨ ¬ *P* (*b*, *c*)
  apply (*induction* - *xs arbitrary*: *i ys rule*: *list-two-induct*)
    apply *fastforce*
   apply *fastforce*
  subgoal for - - - - *ys*
    apply (*cases ys*)
     apply *fastforce*
    subgoal for - *list*
      apply (*cases list*)
       apply *fastforce*+
      done
    done
  done

lemma *successive-successors′*:
  xs = *ys* @ *a* # *b* # *zs* ⟹ *successive P xs* ⟹ ¬ *P a* ∨ ¬ *P b*
using *successive-split* **by** *fastforce*

lemma *cycle-rotate-len-arcs-successive′*:
  fixes *M* :: (′*a* :: *linordered-ab-monoid-add*) *mat*
  assumes *length xs* > *1* (*i*, *j*) ∈ *arcs′ xs successive P* (*arcs a a xs*)
          ¬ *P* (*a*, *hd xs*) ∨ ¬ *P* (*last xs*, *a*)
  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*
                ∧ *successive P* (*arcs i i* (*j* # *ys* @ *a* # *zs*) @ [(*i*,*j*)])
using *assms*
proof −
  note *A* = *assms*
  from *arcs′-decomp*[*OF A*(*1*,*2*)] **obtain** *ys zs* **where** *xs*: *xs* = *zs* @ *i* # *j* # *ys* **by** *blast*
  note *arcs1* = *arcs-decomp*[*OF xs*, *of a a*]
  note *arcs2* = *arcs-decomp*[*of j* # *ys* @ *a* # *zs j* # *ys a zs i i*]
  have ∗:*successive P* (*arcs i i* (*j* # *ys* @ *a* # *zs*) @ [(*i*,*j*)])
  proof (*cases ys* = [])
    case *True*
    show *?thesis*
    proof (*cases zs*)
      case *Nil*
      with *A*(*3*,*4*) *xs True* **show** *?thesis* **by** *auto*
    next

case (*Cons z zs'*)
with *True arcs2 A*(*3*,*4*) *xs* **show** *?thesis*
apply *simp*
apply (*cases P* (*a*, *z*))
apply (*simp add*: *arcs-decomp*)
using *successive-split*[*of* ((*a*, *z*) # *arcs z i zs'*) @ [(*i*, *j*), (*j*, *a*)] - [(*j*, *a*)] *P*]
apply *auto*[]
by (*metis append-Cons arcs.simps*(*1*,*2*) *arcs1 successive.simps*(*1*) *successive-dest-tail successive-ends-1 successive-step*)
qed
next
case *False*
then obtain *y ys'* **where** *ys*: *ys* = *ys'* @ [*y*] **by** (*metis append-butlast-last-id*)
show *?thesis*
proof (*cases zs*)
case *Nil*
with *A*(*3*,*4*) *xs ys* **have** ∗:
¬ *P* (*a*, *i*) ∨ ¬ *P* (*y*, *a*) *successive P* (*arcs a a* (*i* # *j* # *ys'* @ [*y*]))
by *simp*+
from *successive-decomp*[*OF* - *this*(*2*,*1*)] *ys Nil arcs-decomp* **have**
successive P (*arcs i i* (*j* # *ys* @ *a* # *zs*))
by *fastforce*
moreover from ∗ **have** ¬ *P* (*a*, *i*) ∨ ¬ *P* (*i*, *j*) **by** *auto*
ultimately show *?thesis*
by (*metis append-Cons last-snoc list.distinct*(*1*) *list.sel*(*1*) *Nil successive-arcs-extend-last*)
next
case (*Cons z zs'*)
with *A*(*3*,*4*) *xs ys* **have** ∗:
¬ *P* (*a*, *z*) ∨ ¬ *P* (*y*, *a*) *successive P* (*arcs a a* (*z* # *zs'* @ *i* # *j* # *ys'* @ [*y*]))
by *simp*+
from *successive-decomp*[*OF* - *this*(*2*,*1*)] *ys Cons arcs-decomp* **have** ∗∗:
successive P (*arcs i i* (*j* # *ys* @ *a* # *zs*))
by *fastforce*
from *Cons* **have** *zs* ≠ [] **by** *auto*
then obtain *w ws* **where** *ws*: *zs* = *ws* @ [*w*] **by** (*induction zs*) *auto*
with *A*(*3*,*4*) *xs ys* **have** ∗:
successive P (*arcs a a* (*ws* @ [*w*] @ *i* # *j* # *ys'* @ [*y*]))
by *simp*
moreover from *successive-successors*[*OF* - *this*] **have** ¬ *P* (*w*, *i*) ∨ ¬ *P* (*i*, *j*) **by** *auto*

ultimately show *?thesis*
by (*metis* ∗∗ *append-is-Nil-conv last.simps last-append list.distinct*(*2*) *list.sel*(*1*)
successive-arcs-extend-last ws)
qed
qed
from *len-decomp*[*OF xs*, *of M a a*] **have** *len M a a xs* = *len M a i zs* + *len M i a* (*j* # *ys*) **.**
also have ... = *len M i a* (*j* # *ys*) + *len M a i zs* **by** (*simp add*: *comm*)
also from *len-comp*[*of M i i j* # *ys a zs*] **have** ... = *len M i i* (*j* # *ys* @ *a* # *zs*) **by** *auto*
finally show *?thesis*
using ∗ *xs arcs-decomp*[*OF xs*, *of a a*] *arcs-decomp*[*of j* # *ys* @ *a* # *zs j* # *ys a zs i i*] **by** *force*
qed

lemma *cycle-rotate-3*:
fixes *M* :: (*'a* :: *linordered-ab-monoid-add*) *mat*
assumes *xs* ≠ [] (*i*, *j*) ∈ *set* (*arcs a a xs*) *successive P* (*arcs a a xs*) ¬ *P* (*a*, *hd xs*) ∨ ¬ *P* (*last xs*, *a*)
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*))
∧ *successive P* (*arcs i i* (*j* # *ys*))
proof −
note *A* = *assms*
{ fix *ys* **assume** *A*:*a* = *i xs* = *j* # *ys*
with *assms*(*3*) **have** *?thesis* **by** *auto*
} note ∗ = *this*
have ∗∗: *?thesis* **if** *A*: *a* = *j xs* = *ys* @ [*i*] **for** *ys* **using** *A*
proof (*safe*, *goal-cases*)
case *1*
have *len M j j* (*ys* @ [*i*]) = *M i j* + *len M j i ys*
using *len-decomp*[*of ys* @ [*i*] *ys i* [] *M j j*] **by** (*auto simp*: *comm*)
moreover have *arcs j j* (*ys* @ [*i*]) = *arcs j i ys* @ [(*i*, *j*)] **using** *arcs-decomp-tail* **by** *auto*
moreover with *assms*(*3*,*4*) *A* **have** *successive P* ((*i*,*j*) # *arcs j i ys*)
apply *simp*
apply (*cases ys*)
apply *simp*
by (*simp*, *metis arcs.simps*(*2*) *calculation*(*2*) *1*(*1*) *successive-split successive-step*)
ultimately show *?case* **by** *auto*
qed

```
  { assume length xs = 1
    then obtain b where xs: xs = [b] by (metis One-nat-def length-0-conv
length-Suc-conv)
    with A(2) have a = i ∧ b = j ∨ a = j ∧ b = i by auto
    then have ?thesis using * ** xs by auto
  } note *** = this
  show ?thesis
  proof (cases length xs = 0)
    case True with A show ?thesis by auto
  next
    case False
    thus ?thesis
    proof (cases length xs = 1, goal-cases)
      case True with *** show ?thesis by auto
    next
      case 2
      hence length xs > 1 by linarith
      then obtain b c ys where ys:xs = b # ys @ [c]
     by (metis One-nat-def assms(1) 2(2) length-0-conv length-Cons list.exhaust
rev-exhaust)
      thus ?thesis
      proof (cases (i,j) = (a,b))
        case True
        with ys * show ?thesis by blast
      next
        case False
        then show ?thesis
        proof (cases (i,j) = (c,a), goal-cases)
          case True
          with ys ** show ?thesis by force
        next
          case 2
          with A(2) ys have (i, j) ∈ arcs' xs
          using cycle-rotate-2-aux by (auto simp add: arcs'-def)
        from cycle-rotate-len-arcs-successive[OF ‹length xs > 1› this A(3,4),
of M] show ?thesis
          by auto
        qed
      qed
    qed
  qed
qed

lemma cycle-rotate-3':
```

fixes *M* :: (*'a* :: *linordered-ab-monoid-add*) *mat*
assumes *xs* ≠ [] (*i*, *j*) ∈ *set* (*arcs a a xs*) *successive P* (*arcs a a xs*) ¬ *P* (*a*, *hd xs*) ∨ ¬ *P* (*last xs*, *a*)
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*))
∧ *successive P* (*arcs i i* (*j* # *ys*) @ [(*i*, *j*)])
proof −
note *A* = *assms*
have ∗: *?thesis* **if** *a* = *i xs* = *j* # *ys* **for** *ys*
using *that assms*(*3*) *successive-arcs-extend-last*[*OF assms*(*3*,*4*)] **by** *auto*
have ∗∗: *?thesis* **if** *A*:*a* = *j xs* = *ys* @ [*i*] **for** *ys*
using *A* **proof** (*safe*, *goal-cases*)
case *1*
have *len M j j* (*ys* @ [*i*]) = *M i j* + *len M j i ys*
using *len-decomp*[*of ys* @ [*i*] *ys i* [] *M j j*] **by** (*auto simp*: *comm*)
moreover have *arcs j j* (*ys* @ [*i*]) = *arcs j i ys* @ [(*i*, *j*)] **using** *arcs-decomp-tail* **by** *auto*
moreover with *assms*(*3*,*4*) *A* **have** *successive P* ((*i*,*j*) # *arcs j i ys* @ [(*i*, *j*)])
apply *simp*
apply (*cases ys*)
apply *simp*
by (*simp*, *metis successive-step*)
ultimately show *?case* **by** *auto*
qed
{ assume *length xs* = *1*
then obtain *b* **where** *xs*: *xs* = [*b*] **by** (*metis One-nat-def length-0-conv length-Suc-conv*)
with *A*(*2*) **have** *a* = *i* ∧ *b* = *j* ∨ *a* = *j* ∧ *b* = *i* **by** *auto*
then have *?thesis* **using** ∗ ∗∗ *xs* **by** *auto*
} note ∗∗∗ = *this*
show *?thesis*
proof (*cases length xs* = *0*)
case *True* **with** *A* **show** *?thesis* **by** *auto*
next
case *False*
thus *?thesis*
proof (*cases length xs* = *1*, *goal-cases*)
case *True* **with** ∗∗∗ **show** *?thesis* **by** *auto*
next
case *2*
hence *length xs* > *1* **by** *linarith*
then obtain *b c ys* **where** *ys*:*xs* = *b* # *ys* @ [*c*]

by (*metis One-nat-def assms*(*1*) *2*(*2*) *length-0-conv length-Cons list.exhaust rev-exhaust*)
thus *?thesis*
proof (*cases* (*i*,*j*) = (*a*,*b*))
case *True*
with *ys* ∗ **show** *?thesis* **by** *blast*
next
case *False*
then show *?thesis*
proof (*cases* (*i*,*j*) = (*c*,*a*), *goal-cases*)
case *True*
with *ys* ∗∗ **show** *?thesis* **by** *force*
next
case *2*
with *A*(*2*) *ys* **have** (*i*, *j*) ∈ *arcs′ xs*
using *cycle-rotate-2-aux* **by** (*auto simp add*: *arcs′-def*)
from *cycle-rotate-len-arcs-successive′*[*OF* ‹*length xs* > *1*› *this* *A*(*3*,*4*), *of M*] **show** *?thesis*
by *auto*
qed
qed
qed
qed
qed

end

2.5.2 Zones and DBMs

theory *Zones*
imports *DBM*
begin

type-synonym (*′c*, *′t*) *zone* = (*′c*, *′t*) *cval set*

type-synonym (*′c*, *′t*) *cval* = *′c* ⇒ *′t*

definition *cval-add* :: (*′c*,*′t*) *cval* ⇒ *′t*::*plus* ⇒ (*′c*,*′t*) *cval* (**infixr** ‹⊕› *64*)
where
u ⊕ *d* = (λ *x*. *u x* + *d*)

definition *zone-delay* :: (*′c*, (*′t*::*time*)) *zone* ⇒ (*′c*, *′t*) *zone*
(‹-↑› [*71*] *71*)
where

$Z^{\uparrow} = \{u \oplus d|u\ d.\ u \in Z \wedge d \geq (0::'t)\}$

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
$[r \rightarrow t]u \equiv$ *clock-set r t u*

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

lemma *clock-set-set*[*simp*]:
$([r \rightarrow d]u)\ c = d$ **if** $c \in$ *set r*
using *that* **by** (*induction r*) *auto*

lemma *clock-set-id*[*simp*]:
$([r \rightarrow d]u)\ c = u\ c$ **if** $c \notin$ *set r*
using *that* **by** (*induction r*) *auto*

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-mono1*:
dbm-entry-val u (*Some c*) (*Some c'*) *b* ⟹ $b \preceq b'$ ⟹ *dbm-entry-val u* (*Some c*) (*Some c'*) *b'*
proof (*induction b*, *goal-cases*)
case *1* **thus** *?case* **using** *le-dbm-le le-dbm-lt* **by** − (*cases b'*; *fastforce*)
next
case *2* **thus** *?case* **using** *lt-dbm-le lt-dbm-lt* **by** (*cases b'*; *fastforce*)
next
case *3* **thus** *?case* **unfolding** *dbm-le-def* **by** *auto*
qed

lemma *dbm-entry-val-mono2*:

dbm-entry-val u None (*Some c*) *b* $\Longrightarrow$ *b* $\preceq$ *b'* $\Longrightarrow$ *dbm-entry-val u None* (*Some c*) *b'*
proof (*induction b*, *goal-cases*)
  case *1* **thus** *?case* **using** *le-dbm-le le-dbm-lt* **by** − (*cases b'*; *fastforce*)
next
  case *2* **thus** *?case* **using** *lt-dbm-le lt-dbm-lt* **by** (*cases b'*; *fastforce*)
next
  case *3* **thus** *?case* **unfolding** *dbm-le-def* **by** *auto*
qed

lemma *dbm-entry-val-mono3*:
  dbm-entry-val u (*Some c*) *None b* $\Longrightarrow$ *b* $\preceq$ *b'* $\Longrightarrow$ *dbm-entry-val u* (*Some c*) *None b'*
proof (*induction b*, *goal-cases*)
  case *1* **thus** *?case* **using** *le-dbm-le le-dbm-lt* **by** − (*cases b'*; *fastforce*)
next
  case *2* **thus** *?case* **using** *lt-dbm-le lt-dbm-lt* **by** (*cases b'*; *fastforce*)
next
  case *3* **thus** *?case* **unfolding** *dbm-le-def* **by** *auto*
qed

lemmas *dbm-entry-val-mono* = *dbm-entry-val-mono1 dbm-entry-val-mono2 dbm-entry-val-mono3*

lemma *DBM-le-subset*:
  $\forall$ *i j*. $i \le n \longrightarrow j \le n \longrightarrow M\ i\ j \preceq M'\ i\ j \Longrightarrow u \in [M]_{v,n} \Longrightarrow u \in [M']_{v,n}$
proof −
  assume *A*: $\forall$ *i j*. $i \le n \longrightarrow j \le n \longrightarrow M\ i\ j \preceq M'\ i\ j$ $u \in [M]_{v,n}$
  hence *DBM-val-bounded v u M n* **by** (*simp add*: *DBM-zone-repr-def*)
  with *A*(*1*) **have** *DBM-val-bounded v u M' n* **unfolding** *DBM-val-bounded-def*
  proof (*safe*, *goal-cases*)
    case *1* **from** *this*(*1*,*2*) **show** *?case* **unfolding** *less-eq*[*symmetric*] **by** *fastforce*
  next
    case (*2 c*)
    hence *dbm-entry-val u None* (*Some c*) (*M 0* (*v c*)) *M 0* (*v c*) $\preceq$ *M' 0* (*v c*) **by** *auto*
    thus *?case* **using** *dbm-entry-val-mono2* **by** *fast*
  next
    case (*3 c*)
    hence *dbm-entry-val u* (*Some c*) *None* (*M* (*v c*) *0*) *M* (*v c*) *0* $\preceq$ *M'* (*v c*) *0* **by** *auto*
    thus *?case* **using** *dbm-entry-val-mono3* **by** *fast*
  next

case (*4 c1 c2*)
hence *dbm-entry-val u* (*Some c1*) (*Some c2*) (*M* (*v c1*) (*v c2*)) *M* (*v c1*) (*v c2*) $\preceq$ *M'* (*v c1*) (*v c2*)
by *auto*
thus *?case* **using** *dbm-entry-val-mono1* **by** *fast*
qed
thus $u \in [M']_{v,n}$ **by** (*simp add*: *DBM-zone-repr-def*)
qed

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

2.5.3 Useful definitions

fun *get-const* **where**
get-const (*Le c*) = *c* |
get-const (*Lt c*) = *c* |
get-const (∞ :: - *DBMEntry*) = *undefined*

2.5.4 Updating DBMs

abbreviation *DBM-update* :: (*'t*::*time*) *DBM* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ (*'t DBMEntry*) $\Rightarrow$ (*'t*::*time*) *DBM*
where
DBM-update M m n v $\equiv$ (λ *x y*. *if m* = *x* $\wedge$ *n* = *y then v else M x y*)

fun *DBM-upd* :: (*'t*::*time*) *DBM* $\Rightarrow$ (*nat* $\Rightarrow$ *nat* $\Rightarrow$ *'t DBMEntry*) $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *'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 $j \leq n$
shows *DBM-upd M1 f* (*Suc m*) *n N* (*Suc m*) *j* = *DBM-upd M1 f* (*Suc m*) *j N* (*Suc m*) *j*

using *assms*
by (*induction n*) *auto*

lemma *upd-2*:
assumes $i \le m$
shows *DBM-upd M1 f* (*Suc m*) *n N i j* = *DBM-upd M1 f* (*Suc m*) *0 N i j*
using *assms*
proof (*induction n*)
  case *0* **thus** *?case* **by** *blast*
next
  case (*Suc n*)
  thus *?case* **by** *simp*
qed

lemma *upd-3*:
assumes $m \le N$ $n \le N$ $j \le n$ $i \le m$
shows (*DBM-upd M1 f m n N*) *i j* = (*DBM-upd M1 f i j N*) *i j*
using *assms*
proof (*induction m arbitrary*: *n i j*, *goal-cases*)
  case (*1 n*) **thus** *?case* **by** (*induction n*) *auto*
next
  case (*2 m n i j*) **thus** *?case*
  proof (*cases i* = *Suc m*)
    case *True* **thus** *?thesis* **using** *upd-1*[*OF* ‹$j \le n$›] **by** *blast*
    next
    case *False*
    with ‹$i \le$ *Suc m*› **have** $i \le m$ **by** *auto*
    with *upd-2*[*OF this*] **have** *DBM-upd M1 f* (*Suc m*) *n N i j* = *DBM-upd M1 f m N N i j* **by** *force*
    also have ... = *DBM-upd M1 f i j N i j* **using** *False 2* **by** *force*
    finally show *?thesis* **.**
  qed
qed

lemma *upd-id*:
  assumes $m \le N$ $n \le N$ $i \le m$ $j \le n$
  shows (*DBM-upd M1 f m n N*) *i j* = *f i j*
proof –
  from *assms upd-3* **have** *DBM-upd M1 f m n N i j* = *DBM-upd M1 f i j N i j* **by** *blast*
  also have ... = *f i j* **by** (*cases i*; *cases j*; *fastforce*)
  finally show *?thesis* **.**
qed

2.5.5 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′*:
 $(a :: (('a :: time)\ DBMEntry)) \geq 0 \Longrightarrow a + b > 0 \Longrightarrow \exists\ d.\ Le\ d \leq a \wedge Le\ (-d) \leq b \wedge d \geq 0$
proof −
 assume $a + b > 0$ $a \geq 0$
 show *?thesis*
 proof (*cases* $b \geq 0$)
  case *True*
  with ‹$a \geq 0$› **show** *?thesis* **by** (*auto simp*: *neutral*)
 next
  case *False*
  hence $b < Le\ 0$ **by** (*auto simp*: *neutral*)
  note ∗ = *this* ‹$a \geq 0$› ‹$a + b > 0$›
  note [*simp*] = *neutral*
  show *?thesis*
  proof (*cases a*, *cases b*, *goal-cases*)
   case ($1\ a'\ b'$)
   with ∗ **have** $a' + b' > 0$ **by** (*auto elim*: *dbm-lt.cases simp*: *less add*)
   hence $b' > -a'$ **by** (*metis add.commute diff-0 diff-less-eq*)
   with ∗ *1* **show** *?case*
    by (*auto simp*: *dbm-le-def less-eq le-dbm-le*)
  next
   case ($2\ a'\ b'$)
   with ∗ **have** $a' + b' > 0$ **by** (*auto elim*: *dbm-lt.cases simp*: *less add*)
   hence $b' > -a'$ **by** (*metis add.commute diff-0 diff-less-eq*)
   with ∗ *2* **show** *?case*
    by (*auto simp*: *dbm-le-def less-eq le-dbm-le*)
  next
   case ($3\ a'$)
   with ∗ **show** *?case*
    by *auto*
  next

case (*4 a'*)
thus *?case*
proof (*cases b*, *goal-cases*)
case (*1 b'*)
have $b' < 0$ **using** *1*(*2*) $*$ **by** (*metis dbm-lt.intros*(*3*) *less less-asym neqE*)
from *1* $*$ **have** $a' + b' > 0$ **by** (*auto elim*: *dbm-lt.cases simp*: *less add*)
then have $-b' < a'$ **by** (*metis diff-0 diff-less-eq*)
with ‹$b' < 0$› $*$ *1* **show** *?case* **by** (*auto simp*: *dbm-le-def less-eq*)
next
case (*2 b'*)
with $*$ **have** *A*: $b' \leq 0$ $a' > 0$ **by** (*auto elim*: *dbm-lt.cases simp*: *less less-eq dbm-le-def*)
show *?case*
proof (*cases* $b' = 0$)
case *True*
from *dense*[*OF A*(*2*)] **obtain** *d* **where** *d*: $d > 0$ $d < a'$ **by** *auto*
then have *Le* $(-d) <$ *Lt* b' *Le* $d <$ *Lt* a' **unfolding** *less* **using** *True* **by** *auto*
with *d*(*1*) *2* $*$ **show** *?thesis* **by** $-$ (*rule exI*[**where** $x = d$], *auto*)
next
case *False*
with *A*(*1*) **have** $**$: $-\ b' > 0$ **by** *simp*
from *2* $*$ **have** $a' + b' > 0$ **by** (*auto elim*: *dbm-lt.cases simp*: *less add*)
then have $-b' < a'$ **by** (*metis less-add-same-cancel1 minus-add-cancel minus-less-iff*)
from *dense*[*OF this*] **obtain** *d* **where** *d*:
$d > -b'$ $-d < b'$ $d < a'$
by (*auto simp add*: *minus-less-iff*)
then have *Le* $(-d) <$ *Lt* b' *Le* $d <$ *Lt* a' **unfolding** *less* **by** *auto*
with *d*(*1*) *2* $**$ **show** *?thesis*
by $-$ (*rule exI*[**where** $x = d$], *auto*,
meson d(*2*) *dual-order.order-iff-strict less-trans neg-le-0-iff-le*)
qed
next
case *3*
with $*$ **show** *?case*
by *auto*
qed
next
case *5* **thus** *?case*
proof (*cases b*, *goal-cases*)

```
      case (1 b')
      with * have -b' ≥ 0
        by (metis dbm-lt.intros(3) leI less less-asym neg-less-0-iff-less)
      let ?d = - b'
      have Le ?d ≤ ∞ Le (- ?d) ≤ Le b' by (auto simp: any-le-inf)
      with ‹-b' ≥ 0› * 1 show ?case by auto
    next
      case (2 b')
      with * have b' ≤ 0 by (auto elim: dbm-lt.cases simp: less)
      from non-trivial-neg obtain e :: 'a where e:e < 0 by blast
      let ?d = - (b' + e)
      from e ‹b' ≤ 0› have Le ?d ≤ ∞ Le (- ?d) ≤ Lt b' b' + e < 0
     by (auto simp: dbm-lt.intros(4) less less-imp-le any-le-inf add-nonpos-neg)
      then have Le ?d ≤ ∞ Le (- ?d) ≤ Lt b' ?d ≥ 0
        using less-imp-le neg-0-le-iff-le by blast+
      with * 2 show ?case by auto
    next
      case 3
      with * show ?case
        by auto
    qed
  qed
 qed
qed

lemma sum-gt-neutral-dest:
 (a :: (('a :: time) DBMEntry)) + b > 0 ⟹ ∃ d. Le d ≤ a ∧ Le (-d) ≤
b
proof -
 assume A: a + b > 0
 then have A': b + a > 0 by (simp add: comm)
 show ?thesis
 proof (cases a ≥ 0)
  case True
  with A sum-gt-neutral-dest' show ?thesis by auto
 next
  case False
  { assume b ≤ 0
    with False have a ≤ 0 b ≤ 0 by auto
    from add-mono[OF this] have a + b ≤ 0 by auto
    with A have False by auto
  }
  then have b ≥ 0 by fastforce
  with sum-gt-neutral-dest'[OF this A'] show ?thesis by auto
```

qed
qed

2.5.6 Negative Cycles in DBMs

lemma *DBM-val-bounded-neg-cycle1*:
fixes *i xs* **assumes**
bounded: *DBM-val-bounded v u M n* **and** *A*:$i \le n$ *set xs* $\subseteq$ $\{0..n\}$ *len M i i xs* < 0 **and**
surj-on: $\forall\ k \le n.\ k > 0 \longrightarrow (\exists\ c.\ v\ c = k)$ **and** *at-most*: $i \neq 0$ *cnt 0 xs* ≤ 1
shows *False*
proof –
from *A*(*1*) *surj-on at-most* **obtain** *c* **where** *c*: *v c* = *i* **by** *auto*
with *DBM-val-bounded-len'3*[*OF bounded at-most*(*2*), *of c c*] *A*(*1*,*2*) *surj-on*
have *bounded*:*dbm-entry-val u* (*Some c*) (*Some c*) (*len M i i xs*) **by** *force*
from *A*(*3*) **have** *len M i i xs* $\prec$ *Le 0* **by** (*simp add*: *neutral less*)
then show *False* **using** *bounded* **by** (*cases rule*: *dbm-lt.cases*) (*auto elim*: *dbm-entry-val.cases*)
qed

lemma *cnt-0-I*:
$x \notin$ *set xs* $\Longrightarrow$ *cnt x xs* = *0*
by (*induction xs*) *auto*

lemma *distinct-cnt*: *distinct xs* $\Longrightarrow$ *cnt x xs* $\le$ *1*
apply (*induction xs*)
apply *simp*
subgoal for *a xs*
using *cnt-0-I* **by** (*cases x* = *a*) *fastforce*+
done

lemma *DBM-val-bounded-neg-cycle*:
fixes *i xs* **assumes**
bounded: *DBM-val-bounded v u M n* **and** *A*:$i \le n$ *set xs* $\subseteq$ $\{0..n\}$ *len M i i xs* < 0 **and**
surj-on: $\forall\ k \le n.\ k > 0 \longrightarrow (\exists\ c.\ v\ c = k)$
shows *False*
proof –
from *negative-len-shortest*[*OF* - *A*(*3*)] **obtain** *j ys* **where** *ys*:
distinct (*j* # *ys*) *len M j j ys* < 0 $j \in$ *set* (*i* # *xs*) *set ys* $\subseteq$ *set xs*
by *blast*
show *False*
proof (*cases ys* = [])

case *True*
show *?thesis*
proof (*cases j = 0*)
case *True*
with ‹*ys* = []› *ys bounded* **show** *False* **unfolding** *DBM-val-bounded-def neutral less-eq*[*symmetric*]
by *auto*
next
case *False*
with ‹*ys* = []› *DBM-val-bounded-neg-cycle1*[*OF bounded - - ys*(*2*) *surj-on*] *ys*(*3*) *A*(*1*,*2*)
show *False* **by** *auto*
qed
next
case *False*
from *distinct-arcs-ex*[*OF - - this*, *of j 0 j*] *ys*(*1*) **obtain** *a b* **where** *arc*:
a ≠ *0* (*a*, *b*) ∈ *set* (*arcs j j ys*)
by *auto*
from *cycle-rotate-2'*[*OF False this*(*2*)] **obtain** *zs* **where** *zs*:
len M j j ys = *len M a a* (*b* # *zs*) *set* (*a* # *b* # *zs*) = *set* (*j* # *ys*)
1 + *length zs* = *length ys set* (*arcs j j ys*) = *set* (*arcs a a* (*b* # *zs*))
by *blast*
with *distinct-card*[*OF ys*(*1*)] **have** *distinct* (*a* # *b* # *zs*) **by** (*intro card-distinct*) *auto*
with *distinct-cnt*[*of b* # *zs*] **have** ∗: *cnt 0* (*b* # *zs*) ≤ *1* **by** *fastforce*
show *?thesis*
apply (*rule DBM-val-bounded-neg-cycle1*[*OF bounded - - - surj-on* ‹*a* ≠ *0*› ∗])
using *zs*(*2*) *ys*(*3*,*4*) *A*(*1*,*2*) **apply** *fastforce*+
using *zs*(*1*) *ys*(*2*) **by** *simp*
qed
qed

Nicer Path Boundedness Theorems **lemma** *DBM-val-bounded-len-1*:
fixes *v*
assumes *DBM-val-bounded v u M n v c* ≤ *n set vs* ⊆ {*0..n*} ∀ *k* ≤ *n*. (∃ *c*. *v c* = *k*)
shows *dbm-entry-val u* (*Some c*) *None* (*len M* (*v c*) *0 vs*) **using** *assms*
proof (*induction length vs arbitrary*: *vs rule*: *less-induct*)
case *A*: *less*
show *?case*
proof (*cases 0* ∈ *set vs*)
case *False*

with *DBM-val-bounded-len-1′-aux*[*OF A*(*2*,*3*)] *A*(*4*,*5*) **show** *?thesis* **by** *fastforce*
next
case *True*
then obtain *xs ys* **where** *vs*: *vs* = *xs* @ *0* # *ys* **by** (*meson split-list*)
from *len-decomp*[*OF this*] **have** *len M* (*v c*) *0 vs* = *len M* (*v c*) *0 xs* + *len M 0 0 ys* **.**
moreover have *len M 0 0 ys* ≥ *0*
proof (*rule ccontr*, *goal-cases*)
case *1*
then have *len M 0 0 ys* < *0* **by** *simp*
with *DBM-val-bounded-neg-cycle*[*OF assms*(*1*), *of 0 ys*] *vs A*(*4*,*5*) **show** *False* **by** *auto*
qed
ultimately have ∗: *len M* (*v c*) *0 vs* ≥ *len M* (*v c*) *0 xs* **by** (*simp add*: *add-increasing2*)
from *vs A* **have** *dbm-entry-val u* (*Some c*) *None* (*len M* (*v c*) *0 xs*) **by** *auto*
from *dbm-entry-val-mono3*[*OF this*, *of len M* (*v c*) *0 vs*] ∗ **show** *?thesis* **unfolding** *less-eq* **by** *auto*
qed
qed

lemma *DBM-val-bounded-len-2*:
fixes *v*
assumes *DBM-val-bounded v u M n v c* ≤ *n set vs* ⊆ {*0*..*n*} ∀ *k* ≤ *n*. (∃ *c*. *v c* = *k*)
shows *dbm-entry-val u None* (*Some c*) (*len M 0* (*v c*) *vs*) **using** *assms*
proof (*induction length vs arbitrary*: *vs rule*: *less-induct*)
case *A*: *less*
show *?case*
proof (*cases 0* ∈ *set vs*)
case *False*
with *DBM-val-bounded-len-2′-aux*[*OF A*(*2*,*3*)] *A*(*4*,*5*) **show** *?thesis* **by** *fastforce*
next
case *True*
then obtain *xs ys* **where** *vs*: *vs* = *xs* @ *0* # *ys* **by** (*meson split-list*)
from *len-decomp*[*OF this*] **have** *len M 0* (*v c*) *vs* = *len M 0 0 xs* + *len M 0* (*v c*) *ys* **.**
moreover have *len M 0 0 xs* ≥ *0*
proof (*rule ccontr*, *goal-cases*)
case *1*
then have *len M 0 0 xs* < *0* **by** *simp*

with *DBM-val-bounded-neg-cycle*[*OF assms*(*1*), *of 0 xs*] *vs A*(*4*,*5*)
show *False* **by** *auto*
qed
ultimately have ∗: *len M 0* (*v c*) *vs* ≥ *len M 0* (*v c*) *ys* **by** (*simp add*: *add-increasing*)
from *vs A* **have** *dbm-entry-val u None* (*Some c*) (*len M 0* (*v c*) *ys*) **by** *auto*
from *dbm-entry-val-mono2*[*OF this*] ∗ **show** *?thesis* **unfolding** *less-eq* **by** *auto*
qed
qed

lemma *DBM-val-bounded-len-3*:
fixes *v*
assumes *DBM-val-bounded v u M n v c1* ≤ *n v c2* ≤ *n set vs* ⊆ {*0..n*}
∀ *k* ≤ *n*. (∃ *c*. *v c* = *k*)
shows *dbm-entry-val u* (*Some c1*) (*Some c2*) (*len M* (*v c1*) (*v c2*) *vs*)
using *assms*
proof (*cases 0* ∈ *set vs*)
case *False*
with *DBM-val-bounded-len-3′-aux*[*OF assms*(*1*−*3*)] *assms*(*4*−) **show** *?thesis* **by** *fastforce*
next
case *True*
then obtain *xs ys* **where** *vs*: *vs* = *xs* @ *0* # *ys* **by** (*meson split-list*)
from *assms*(*4*,*5*) *vs DBM-val-bounded-len-1*[*OF assms*(*1*,*2*)] *DBM-val-bounded-len-2*[*OF assms*(*1*,*3*)]
have
dbm-entry-val u (*Some c1*) *None* (*len M* (*v c1*) *0 xs*)
dbm-entry-val u None (*Some c2*) (*len M 0* (*v c2*) *ys*)
by *auto*
from *dbm-entry-val-add-4*[*OF this*] *len-decomp*[*OF vs*, *of M*] **show** *?thesis* **unfolding** *add* **by** *auto*
qed

An equivalent way of handling **0**

fun *val-0* :: (′*c* ⇒ (′*a* :: *linordered-ab-group-add*)) ⇒ ′*c option* ⇒ ′*a* **where**
val-0 u None = *0* |
val-0 u (*Some c*) = *u c*

notation *val-0* (‹-**₀** -› [*90*,*90*] *90*)

lemma *dbm-entry-val-None-None*[*dest*]:
dbm-entry-val u None None l ⟹ *l* = ∞

by (*auto elim*: *dbm-entry-val.cases*)

lemma *dbm-entry-val-dbm-lt*:
  assumes *dbm-entry-val u x y l*
  shows $Lt\ (u_0\ x - u_0\ y) \prec l$
using *assms* **by** (*cases rule*: *dbm-entry-val.cases*, *auto*)

lemma *dbm-lt-dbm-entry-val-1*:
  assumes $Lt\ (u\ x) \prec l$
  shows *dbm-entry-val u* (*Some x*) *None l*
using *assms* **by** (*cases rule*: *dbm-lt.cases*) *auto*

lemma *dbm-lt-dbm-entry-val-2*:
  assumes $Lt\ (-\ u\ x) \prec l$
  shows *dbm-entry-val u None* (*Some x*) *l*
using *assms* **by** (*cases rule*: *dbm-lt.cases*) *auto*

lemma *dbm-lt-dbm-entry-val-3*:
  assumes $Lt\ (u\ x - u\ y) \prec l$
  shows *dbm-entry-val u* (*Some x*) (*Some y*) *l*
using *assms* **by** (*cases rule*: *dbm-lt.cases*) *auto*

A more uniform theorem for boundedness by paths

lemma *DBM-val-bounded-len*:
  fixes *v*
  defines $v' \equiv \lambda\ x.\ if\ x = None\ then\ 0\ else\ v\ (the\ x)$
  assumes $DBM\text{-}val\text{-}bounded\ v\ u\ M\ n\ v'\ x \le n\ v'\ y \le n\ set\ vs \subseteq \{0..n\}$
    $\forall\ k \le n.\ (\exists\ c.\ v\ c = k)\ x \neq None \lor y \neq None$
  shows $Lt\ (u_0\ x - u_0\ y) \prec len\ M\ (v'\ x)\ (v'\ y)\ vs$ **using** *assms*
apply −
apply (*rule dbm-entry-val-dbm-lt*)
apply (*cases x*; *cases y*)
   apply *simp-all*
  apply (*rule DBM-val-bounded-len-2*; *auto*)
 apply (*rule DBM-val-bounded-len-1*; *auto*)
apply (*rule DBM-val-bounded-len-3*; *auto*)
done

2.5.7 Floyd-Warshall Algorithm Preservers Zones

lemma *D-dest*: $x = D\ m\ i\ j\ k \Longrightarrow$
  $x \in \{len\ m\ i\ j\ xs\ |xs.\ set\ xs \subseteq \{0..k\} \land i \notin set\ xs \land j \notin set\ xs \land distinct\ xs\}$
using *Min-elem-dest*[*OF D-base-finite″ D-base-not-empty*] **by** (*fastforce simp*

add: *D-def*)

lemma *FW-zone-equiv*:
$\forall\ k \le n.\ k > 0 \longrightarrow (\exists\ c.\ v\ c = k) \Longrightarrow [M]_{v,n} = [FW\ M\ n]_{v,n}$
proof *safe*
fix *u* **assume** *A*: $u \in [FW\ M\ n]_{v,n}$
{ **fix** *i j* **assume** $i \le n$ $j \le n$
hence *FW M n i j* $\le$ *M i j* **using** *fw-mono*[*of i n j M*] **by** *simp*
hence *FW M n i j* $\preceq$ *M i j* **by** (*simp add*: *less-eq*)
}
with *DBM-le-subset*[*of n FW M n M*] *A* **show** $u \in [M]_{v,n}$ **by** *auto*
next
fix *u* **assume** *u*:$u \in [M]_{v,n}$ **and** *surj-on*: $\forall\ k \le n.\ k > 0 \longrightarrow (\exists\ c.\ v\ c = k)$
hence $*$:*DBM-val-bounded v u M n* **by** (*simp add*: *DBM-zone-repr-def*)
note $** =$ *DBM-val-bounded-neg-cycle*[*OF this - - - surj-on*]
have *cyc-free*: *cyc-free M n* **using** $**$ **by** *fastforce*
from *cyc-free-diag*[*OF this*] **have** *diag-ge-zero*: $\forall k \le n.\ M\ k\ k \ge Le\ 0$ **unfolding** *neutral* **by** *auto*

have *DBM-val-bounded v u* (*FW M n*) *n* **unfolding** *DBM-val-bounded-def*
proof (*safe*, *goal-cases*)
case *1*
from *fw-shortest-path*[*OF cyc-free*] **have** $**$:
D M 0 0 n = *FW M n 0 0*
by (*simp add*: *neutral*)
from *D-dest*[*OF* $**$[*symmetric*]] **obtain** *xs* **where** *xs*:
FW M n 0 0 = *len M 0 0 xs set xs* $\subseteq \{0..n\}$
$0 \notin$ *set xs distinct xs*
by *auto*
with *cyc-free* **have** *FW M n 0 0* ≥ 0 **by** *auto*
then show *?case* **unfolding** *neutral less-eq* **by** *simp*
next
case (*2 c*)
with *fw-shortest-path*[*OF cyc-free*] **have** $**$:
D M 0 (*v c*) *n* = *FW M n 0* (*v c*)
by (*simp add*: *neutral*)
from *D-dest*[*OF* $**$[*symmetric*]] **obtain** *xs* **where** *xs*:
FW M n 0 (*v c*) = *len M 0* (*v c*) *xs set xs* $\subseteq \{0..n\}$
$0 \notin$ *set xs v c* $\notin$ *set xs distinct xs*
by *auto*
show *?case* **unfolding** *xs*(*1*) **using** *xs surj-on* ‹$v\ c \le n$›
by − (*rule DBM-val-bounded-len'2*[*OF* $*$ *xs*(*3*)]; *auto*)
next

case (*3 c*)
with *fw-shortest-path*[*OF cyc-free*] **have** ∗∗:
D M (*v c*) *0 n* = *FW M n* (*v c*) *0*
by (*simp add*: *neutral*)
with *D-dest*[*OF* ∗∗[*symmetric*]] **obtain** *xs* **where** *xs*:
FW M n (*v c*) *0* = *len M* (*v c*) *0 xs set xs* ⊆ {*0*..*n*}
0 ∉ *set xs v c* ∉ *set xs distinct xs*
by *auto*
show *?case* **unfolding** *xs*(*1*) **using** *xs surj-on* ‹*v c* ≤ *n*›
by − (*rule DBM-val-bounded-len′1*[*OF* ∗ *xs*(*3*)]; *auto*)
next
case (*4 c1 c2*)
with *fw-shortest-path*[*OF cyc-free*]
have *D M* (*v c1*) (*v c2*) *n* = *FW M n* (*v c1*) (*v c2*) **by** (*simp add*: *neutral*)
from *D-dest*[*OF this*[*symmetric*]] **obtain** *xs* **where** *xs*:
FW M n (*v c1*) (*v c2*) = *len M* (*v c1*) (*v c2*) *xs set xs* ⊆ {*0*..*n*}
v c1 ∉ *set xs v c2* ∉ *set xs distinct xs*
by *auto*
show *?case*
unfolding *xs*(*1*)
apply (*rule DBM-val-bounded-len′3*[*OF* ∗])
using *xs surj-on* ‹*v c1* ≤ *n*› ‹*v c2* ≤ *n*› **by** (*auto dest*!: *distinct-cnt*[*of* - *0*])
qed
then show $u \in [FW\ M\ n]_{v,n}$ **unfolding** *DBM-zone-repr-def* **by** *simp*
qed

lemma *new-negative-cycle-aux′*:
fixes *M* :: (*′a* :: *time*) *DBM*
fixes *i j d*
defines *M′* ≡ λ *i′ j′*. *if* (*i′* = *i* ∧ *j′* = *j*) *then Le d*
else if (*i′* = *j* ∧ *j′* = *i*) *then Le* (−*d*)
else M i′ j′
assumes *i* ≤ *n j* ≤ *n set xs* ⊆ {*0*..*n*} *cycle-free M n length xs* = *m*
assumes *len M′ i i* (*j* # *xs*) < *0* ∨ *len M′ j j* (*i* # *xs*) < *0*
assumes *i* ≠ *j*
shows ∃ *xs*. *set xs* ⊆ {*0*..*n*} ∧ *j* ∉ *set xs* ∧ *i* ∉ *set xs*
∧ (*len M′ i i* (*j* # *xs*) < *0* ∨ *len M′ j j* (*i* # *xs*) < *0*) **using** *assms*
proof (*induction* - *m arbitrary*: *xs rule*: *less-induct*)
case (*less x*)
{ fix *b a xs* **assume** *A*: (*i*, *j*) ∉ *set* (*arcs b a xs*) (*j*, *i*) ∉ *set* (*arcs b a xs*)
with ‹*i* ≠ *j*› **have** *len M′ b a xs* = *len M b a xs*

unfolding *M′-def* **by** (*induction xs arbitrary*: *b*) *auto*
} **note** ∗ = *this*
{ **fix** *a xs* **assume** *A*:(*i*, *j*) ∉ *set* (*arcs a a xs*) (*j*, *i*) ∉ *set* (*arcs a a xs*)
assume *a*: *a* ≤ *n* **and** *xs*: *set xs* ⊆ {*0..n*} **and** *cycle*: ¬ *len M′ a a xs* ≥ *0*
from ∗[*OF A*] **have** *len M′ a a xs* = *len M a a xs* **.**
with ‹*cycle-free M n*› ‹*i* ≤ *n*› *cycle xs a* **have** *False* **unfolding** *cycle-free-def* **by** *auto*
} **note** ∗∗ = *this*
{ **fix** *a* :: *nat* **fix** *ys* :: *nat list*
assume *A*: *ys* ≠ [] *length ys* ≤ *length xs set ys* ⊆ *set xs a* ≤ *n*
assume *cycle*: *len M′ a a ys* < *0*
assume *arcs*: (*i*, *j*) ∈ *set* (*arcs a a ys*) ∨ (*j*, *i*) ∈ *set* (*arcs a a ys*)
from *arcs* **have** *?thesis*
proof
assume (*i*, *j*) ∈ *set* (*arcs a a ys*)
from *cycle-rotate-2*[*OF* ‹*ys* ≠ []› *this*, *of M′*]
obtain *ws* **where** *ws*: *len M′ a a ys* = *len M′ i i* (*j* # *ws*) *set ws* ⊆ *set* (*a* # *ys*)
length ws < *length ys* **by** *auto*
with *cycle less.hyps*(*1*)[*OF* - *less.hyps*(*2*) , *of length ws ws*] *less.prems A*
show *?thesis* **by** *fastforce*
next
assume (*j*, *i*) ∈ *set* (*arcs a a ys*)
from *cycle-rotate-2*[*OF* ‹*ys* ≠ []› *this*, *of M′*]
obtain *ws* **where** *ws*: *len M′ a a ys* = *len M′ j j* (*i* # *ws*) *set ws* ⊆ *set* (*a* # *ys*)
length ws < *length ys* **by** *auto*
with *cycle less.hyps*(*1*)[*OF* - *less.hyps*(*2*) , *of length ws ws*] *less.prems A*
show *?thesis* **by** *fastforce*
qed
} **note** ∗∗∗ = *this*
{ **fix** *a* :: *nat* **fix** *ys* :: *nat list*
assume *A*: *ys* ≠ [] *length ys* ≤ *length xs set ys* ⊆ *set xs a* ≤ *n*
assume *cycle*: ¬ *len M′ a a ys* ≥ *0*
with *A* ∗∗[*of a ys*] *less.prems*
have (*i*, *j*) ∈ *set* (*arcs a a ys*) ∨ (*j*, *i*) ∈ *set* (*arcs a a ys*) **by** *auto*
with ∗∗∗[*OF A*] *cycle* **have** *?thesis* **by** *auto*
} **note** *neg-cycle-IH* = *this*
from *cycle-free-diag*[*OF* ‹*cycle-free M n*›] **have** ∀ *i*. *i* ≤ *n* ⟶ *Le 0* ≤ *M i i* **unfolding** *neutral* **by** *auto*
then have *M′-diag*: ∀ *i*. *i* ≤ *n* ⟶ *Le 0* ≤ *M′ i i* **unfolding** *M′-def*

using ‹$i \neq j$› **by** *auto*
  from *less*(*8*) **show** *?thesis*
  proof *standard*
    assume *cycle*:*len* M' *i i* (*j* # *xs*) < *0*
    show *?thesis*
    proof (*cases i* ∈ *set xs*)
      case *False*
      then show *?thesis*
      proof (*cases j* ∈ *set xs*)
        case *False*
        with ‹*i* ∉ *set xs*› **show** *?thesis* **using** *less.prems*(*3*,*6*) **by** *auto*
      next
        case *True*
      then obtain *ys zs* **where** *ys-zs*: *xs* = *ys* @ *j* # *zs* **by** (*meson split-list*)
        with *len-decomp*[*of j* # *xs j* # *ys j zs* M' *i i*]
        have *len*: *len* M' *i i* (*j* # *xs*) = M' *i j* + *len* M' *j j ys* + *len* M' *j i*
zs **by** *auto*
        show *?thesis*
        proof (*cases len* M' *j j ys* ≥ *0*)
          case *True*
          have *len* M' *i i* (*j* # *zs*) = M' *i j* + *len* M' *j i zs* **by** *simp*
          also from *len True* **have** M' *i j* + *len* M' *j i zs* ≤ *len* M' *i i* (*j* #
xs)
          by (*metis add-le-impl add-lt-neutral comm not-le*)
          finally have *cycle'*: *len* M' *i i* (*j* # *zs*) < *0* **using** *cycle* **by** *auto*
          from *ys-zs less.prems*(*5*) **have** *x* > *length zs* **by** *auto*
          from *cycle' less.prems ys-zs less.hyps*(*1*)[*OF this less.hyps*(*2*) , *of*
zs]
          show *?thesis* **by** *auto*
        next
          case *False*
          with *M'-diag less.prems* **have** *ys* ≠ [] **by** (*auto simp*: *neutral*)
        from *neg-cycle-IH*[*OF this*] *ys-zs False less.prems*(*1*,*2*) **show** *?thesis*
by *auto*
        qed
      qed
    next
      case *True*
     then obtain *ys zs* **where** *ys-zs*: *xs* = *ys* @ *i* # *zs* **by** (*meson split-list*)
      with *len-decomp*[*of j* # *xs j* # *ys i zs* M' *i i*]
      have *len*: *len* M' *i i* (*j* # *xs*) = M' *i j* + *len* M' *j i ys* + *len* M' *i i zs*
by *auto*
      show *?thesis*
      proof (*cases len* M' *i i zs* ≥ *0*)

```
        case True
        have len M' i i (j # ys) = M' i j + len M' j i ys by simp
        also from len True have M' i j + len M' j i ys ≤ len M' i i (j #
xs)
        by (metis add-lt-neutral comm not-le)
        finally have cycle': len M' i i (j # ys) < 0 using cycle by auto
        from ys-zs less.prems(5) have x > length ys by auto
        from cycle' less.prems ys-zs less.hyps(1)[OF this less.hyps(2) , of ys]
        show ?thesis by auto
      next
        case False
        with less.prems(1,7) M'-diag have zs ≠ [] by (auto simp: neutral)
        from neg-cycle-IH[OF this] ys-zs False less.prems(1,2) show ?thesis
by auto
      qed
    qed
  next
    assume cycle:len M' j j (i # xs) < 0
    show ?thesis
    proof (cases j ∈ set xs)
      case False
      then show ?thesis
      proof (cases i ∈ set xs)
        case False
        with ‹j ∉ set xs› show ?thesis using less.prems(3,6) by auto
      next
        case True
        then obtain ys zs where ys-zs: xs = ys @ i # zs by (meson split-list)
        with len-decomp[of i # xs i # ys i zs M' j j]
        have len: len M' j j (i # xs) = M' j i + len M' i i ys + len M' i j
zs by auto
        show ?thesis
        proof (cases len M' i i ys ≥ 0)
          case True
          have len M' j j (i # zs) = M' j i + len M' i j zs by simp
          also from len True have M' j i + len M' i j zs ≤ len M' j j (i #
xs)
          by (metis add-le-impl add-lt-neutral comm not-le)
          finally have cycle': len M' j j (i # zs) < 0 using cycle by auto
          from ys-zs less.prems(5) have x > length zs by auto
          from cycle' less.prems ys-zs less.hyps(1)[OF this less.hyps(2) , of
zs]
          show ?thesis by auto
        next
```

```
        case False
        with less.prems M'-diag have ys ≠ [] by (auto simp: neutral)
       from neg-cycle-IH[OF this] ys-zs False less.prems(1,2) show ?thesis
by auto
      qed
    qed
  next
    case True
   then obtain ys zs where ys-zs: xs = ys @ j # zs by (meson split-list)
    with len-decomp[of i # xs i # ys j zs M' j j]
    have len: len M' j j (i # xs) = M' j i + len M' i j ys + len M' j j zs
by auto
    show ?thesis
    proof (cases len M' j j zs ≥ 0)
      case True
      have len M' j j (i # ys) = M' j i + len M' i j ys by simp
      also from len True have M' j i + len M' i j ys ≤ len M' j j (i #
xs)
      by (metis add-lt-neutral comm not-le)
      finally have cycle': len M' j j (i # ys) < 0 using cycle by auto
      from ys-zs less.prems(5) have x > length ys by auto
     from cycle' less.prems ys-zs less.hyps(1)[OF this less.hyps(2) , of ys]
      show ?thesis by auto
    next
      case False
      with less.prems(2,7) M'-diag have zs ≠ [] by (auto simp: neutral)
     from neg-cycle-IH[OF this] ys-zs False less.prems(1,2) show ?thesis
by auto
    qed
  qed
 qed
qed

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) < 0 ∨ len M' i i (0 # xs) < 0
 assumes i ≠ 0
 shows ∃ xs. set xs ⊆ {0..n} ∧ 0 ∉ set xs ∧ i ∉ set xs
          ∧ (len M' 0 0 (i # xs) < 0 ∨ len M' i i (0 # xs) < 0) using
```

assms
proof (*induction - m arbitrary*: *xs rule*: *less-induct*)
  case (*less x*)
  { fix *b a xs* **assume** *A*: (*0*, *i*) $\notin$ *set* (*arcs b a xs*) (*i*, *0*) $\notin$ *set* (*arcs b a xs*)
    then have *len M' b a xs* = *len M b a xs*
    unfolding *M'-def* **by** (*induction xs arbitrary*: *b*) *auto*
  } note $*$ = *this*
  { fix *a xs* **assume** *A*:(*0*, *i*) $\notin$ *set* (*arcs a a xs*) (*i*, *0*) $\notin$ *set* (*arcs a a xs*)
    assume *a*: *a* $\leq$ *n* **and** *xs*: *set xs* $\subseteq$ {*0..n*} **and** *cycle*: $\neg$ *len M' a a xs* $\geq$ *0*
    from $*$[*OF A*] **have** *len M' a a xs* = *len M a a xs* **.**
    with ‹*cycle-free M n*› ‹*i* $\leq$ *n*› *cycle xs a* **have** *False* **unfolding** *cycle-free-def* **by** *auto*
  } note $**$ = *this*
  { fix *a* :: *nat* **fix** *ys* :: *nat list*
    assume *A*: *ys* $\neq$ [] *length ys* $\leq$ *length xs set ys* $\subseteq$ *set xs a* $\leq$ *n*
    assume *cycle*: *len M' a a ys* < *0*
    assume *arcs*: (*0*, *i*) $\in$ *set* (*arcs a a ys*) $\vee$ (*i*, *0*) $\in$ *set* (*arcs a a ys*)
    from *arcs* **have** *?thesis*
    proof
      assume (*0*, *i*) $\in$ *set* (*arcs a a ys*)
      from *cycle-rotate-2*[*OF* ‹*ys* $\neq$ []› *this*, *of M'*]
      obtain *ws* **where** *ws*: *len M' a a ys* = *len M' 0 0* (*i* # *ws*) *set ws* $\subseteq$ *set* (*a* # *ys*)
        length ws < *length ys* **by** *auto*
      with *cycle less.hyps*(*1*)[*OF* - *less.hyps*(*2*) , *of length ws ws*] *less.prems A*
      show *?thesis* **by** *fastforce*
    next
      assume (*i*, *0*) $\in$ *set* (*arcs a a ys*)
      from *cycle-rotate-2*[*OF* ‹*ys* $\neq$ []› *this*, *of M'*]
      obtain *ws* **where** *ws*: *len M' a a ys* = *len M' i i* (*0* # *ws*) *set ws* $\subseteq$ *set* (*a* # *ys*)
        length ws < *length ys* **by** *auto*
      with *cycle less.hyps*(*1*)[*OF* - *less.hyps*(*2*) , *of length ws ws*] *less.prems A*
      show *?thesis* **by** *fastforce*
    qed
  } note $***$ = *this*
  { fix *a* :: *nat* **fix** *ys* :: *nat list*
    assume *A*: *ys* $\neq$ [] *length ys* $\leq$ *length xs set ys* $\subseteq$ *set xs a* $\leq$ *n*
    assume *cycle*: $\neg$ *len M' a a ys* $\geq$ *0*
    with *A* $**$[*of a ys*] *less.prems*(*2*)

have $(0, i) \in set\ (arcs\ a\ a\ ys) \vee (i, 0) \in set\ (arcs\ a\ a\ ys)$ **by** *auto*
with $***[OF\ A]$ *cycle* **have** *?thesis* **by** *auto*
} **note** *neg-cycle-IH* = *this*
from *cycle-free-diag*[*OF* ‹*cycle-free M n*›] **have** $\forall i.\ i \leq n \longrightarrow Le\ 0 \leq M\ i\ i$ **unfolding** *neutral* **by** *auto*
then have *M′-diag*: $\forall i.\ i \leq n \longrightarrow Le\ 0 \leq M'\ i\ i$ **unfolding** *M′-def* **using** ‹$i \neq 0$› **by** *auto*
from *less*(*7*) **show** *?thesis*
proof *standard*
assume *cycle*:$len\ M'\ 0\ 0\ (i\ \#\ xs) < 0$
show *?thesis*
proof (*cases* $0 \in set\ xs$)
case *False*
thus *?thesis*
proof (*cases* $i \in set\ xs$)
case *False*
with ‹$0 \notin set\ xs$› **show** *?thesis* **using** *less.prems* **by** *auto*
next
case *True*
then obtain *ys zs* **where** *ys-zs*: $xs = ys\ @\ i\ \#\ zs$ **by** (*meson split-list*)
with *len-decomp*[*of* $i\ \#\ xs\ i\ \#\ ys\ i\ zs\ M'\ 0\ 0$]
have *len*: $len\ M'\ 0\ 0\ (i\ \#\ xs) = M'\ 0\ i + len\ M'\ i\ i\ ys + len\ M'\ i\ 0\ zs$ **by** *auto*
show *?thesis*
proof (*cases* $len\ M'\ i\ i\ ys \geq 0$)
case *True*
have $len\ M'\ 0\ 0\ (i\ \#\ zs) = M'\ 0\ i + len\ M'\ i\ 0\ zs$ **by** *simp*
also from *len True* **have** $M'\ 0\ i + len\ M'\ i\ 0\ zs \leq len\ M'\ 0\ 0\ (i\ \#\ xs)$
by (*metis add-le-impl add-lt-neutral comm not-le*)
finally have *cycle′*: $len\ M'\ 0\ 0\ (i\ \#\ zs) < 0$ **using** *cycle* **by** *auto*
from *ys-zs less.prems*(*4*) **have** $x > length\ zs$ **by** *auto*
from *cycle′ less.prems ys-zs less.hyps*(*1*)[*OF this less.hyps*(*2*) , *of zs*]
show *?thesis* **by** *auto*
next
case *False*
with *less.prems*(*1*,*6*) *M′-diag* **have** $ys \neq []$ **by** (*auto simp*: *neutral*)
from *neg-cycle-IH*[*OF this*] *ys-zs False less.prems*(*1*,*2*) **show** *?thesis* **by** *auto*
qed
qed
next
case *True*

then obtain *ys zs* **where** *ys-zs*: *xs* = *ys* @ *0* # *zs* **by** (*meson split-list*)
with *len-decomp*[*of i* # *xs i* # *ys 0 zs M′ 0 0*]
have *len*: *len M′ 0 0* (*i* # *xs*) = *M′ 0 i* + *len M′ i 0 ys* + *len M′ 0 0 zs* **by** *auto*
show *?thesis*
proof (*cases len M′ 0 0 zs* ≥ *0*)
case *True*
have *len M′ 0 0* (*i* # *ys*) = *M′ 0 i* + *len M′ i 0 ys* **by** *simp*
also from *len True* **have** *M′ 0 i* + *len M′ i 0 ys* ≤ *len M′ 0 0* (*i* # *xs*)
by (*metis add-lt-neutral comm not-le*)
finally have *cycle′*: *len M′ 0 0* (*i* # *ys*) < *0* **using** *cycle* **by** *auto*
from *ys-zs less.prems*(*4*) **have** *x* > *length ys* **by** *auto*
from *cycle′ less.prems ys-zs less.hyps*(*1*)[*OF this less.hyps*(*2*) , *of ys*]
show *?thesis* **by** *auto*
next
case *False*
with *less.prems*(*1*,*6*) *M′-diag* **have** *zs* ≠ [] **by** (*auto simp*: *neutral*)
from *neg-cycle-IH*[*OF this*] *ys-zs False less.prems*(*1*,*2*) **show** *?thesis* **by** *auto*
qed
qed
next
assume *cycle*: *len M′ i i* (*0* # *xs*) < *0*
show *?thesis*
proof (*cases i* ∈ *set xs*)
case *False*
thus *?thesis*
proof (*cases 0* ∈ *set xs*)
case *False*
with ‹*i* ∉ *set xs*› **show** *?thesis* **using** *less.prems* **by** *auto*
next
case *True*
then obtain *ys zs* **where** *ys-zs*: *xs* = *ys* @ *0* # *zs* **by** (*meson split-list*)
with *len-decomp*[*of 0* # *xs 0* # *ys 0 zs M′ i i*]
have *len*: *len M′ i i* (*0* # *xs*) = *M′ i 0* + *len M′ 0 0 ys* + *len M′ 0 i zs* **by** *auto*
show *?thesis*
proof (*cases len M′ 0 0 ys* ≥ *0*)
case *True*
have *len M′ i i* (*0* # *zs*) = *M′ i 0* + *len M′ 0 i zs* **by** *simp*
also from *len True* **have** *M′ i 0* + *len M′ 0 i zs* ≤ *len M′ i i* (*0* # *xs*)

by (*metis add-le-impl add-lt-neutral comm not-le*)
finally have *cycle'*: *len M' i i* (*0* # *zs*) < *0* **using** *cycle* **by** *auto*
from *ys-zs less.prems*(*4*) **have** *x* > *length zs* **by** *auto*
from *cycle' less.prems ys-zs less.hyps*(*1*)[*OF this less.hyps*(*2*) , *of zs*]
show *?thesis* **by** *auto*
next
case *False*
with *less.prems*(*1*,*6*) *M'-diag* **have** *ys* ≠ [] **by** (*auto simp*: *neutral*)
from *neg-cycle-IH*[*OF this*] *ys-zs False less.prems*(*1*,*2*) **show** *?thesis* **by** *auto*
qed
qed
next
case *True*
then obtain *ys zs* **where** *ys-zs*: *xs* = *ys* @ *i* # *zs* **by** (*meson split-list*)
with *len-decomp*[*of 0* # *xs 0* # *ys i zs M' i i*]
have *len*: *len M' i i* (*0* # *xs*) = *M' i 0* + *len M' 0 i ys* + *len M' i i zs* **by** *auto*
show *?thesis*
proof (*cases len M' i i zs* ≥ *0*)
case *True*
have *len M' i i* (*0* # *ys*) = *M' i 0* + *len M' 0 i ys* **by** *simp*
also from *len True* **have** *M' i 0* + *len M' 0 i ys* ≤ *len M' i i* (*0* # *xs*)
by (*metis add-lt-neutral comm not-le*)
finally have *cycle'*: *len M' i i* (*0* # *ys*) < *0* **using** *cycle* **by** *auto*
from *ys-zs less.prems*(*4*) **have** *x* > *length ys* **by** *auto*
from *cycle' less.prems ys-zs less.hyps*(*1*)[*OF this less.hyps*(*2*) , *of ys*]
show *?thesis* **by** *auto*
next
case *False*
with *less.prems*(*1*,*6*) *M'-diag* **have** *zs* ≠ [] **by** (*auto simp*: *neutral*)
from *neg-cycle-IH*[*OF this*] *ys-zs False less.prems*(*1*,*2*) **show** *?thesis* **by** *auto*
qed
qed
qed
qed

2.6 The Characteristic Property of Canonical DBMs

theorem *fix-index'*:
fixes *M* :: ((*'a* :: *time*) *DBMEntry*) *mat*

assumes *Le r* $\leq$ *M i j Le* $(-r) \leq$ *M j i cycle-free M n canonical M n i* $\leq$ *n j* $\leq$ *n i* $\neq$ *j*
defines $M' \equiv \lambda\ i'\ j'.$ *if* $(i' = i \wedge j' = j)$ *then Le r*
else if $(i' = j \wedge j' = i)$ *then Le* $(-r)$
else M $i'\ j'$
shows ($\forall$ *u*. *DBM-val-bounded v u M' n* $\longrightarrow$ *DBM-val-bounded v u M n*) $\wedge$ *cycle-free M' n*
proof –
note *A* = *assms*
note *r* = *assms*(*1*,*2*)
from ‹*cycle-free M n*› **have** *diag-cycles*: $\forall$ *i xs*. *i* $\leq$ *n* $\wedge$ *set xs* $\subseteq$ {*0..n*} $\longrightarrow$ *Le 0* $\leq$ *len M i i xs*
unfolding *cycle-free-def neutral* **by** *auto*
let *?M'* = $\lambda\ i'\ j'.$ *if* $(i' = i \wedge j' = j)$ *then Le r*
else if $(i' = j \wedge j' = i)$ *then Le* $(-r)$
else M $i'\ j'$
have *?M'* $i'\ j' \leq$ *M* $i'\ j'$ **when** $i' \leq n\ j' \leq n$ **for** $i'\ j'$ **using** *assms* **by** *auto*
with *DBM-le-subset*[*folded less-eq*, *of n ?M' M*] **have** *DBM-val-bounded v u M n*
if *DBM-val-bounded v u ?M' n* **for** *u* **unfolding** *DBM-zone-repr-def* **using** *that* **by** *auto*
then have *not-empty*:$\forall$ *u*. *DBM-val-bounded v u ?M' n* $\longrightarrow$ *DBM-val-bounded v u M n* **by** *auto*
{ fix *a xs* **assume** *prems*: *a* $\leq$ *n set xs* $\subseteq$ {*0..n*} **and** *cycle*: $\neg$ *len ?M' a a xs* $\geq$ *0*
{ fix *b* **assume** *A*: (*i*, *j*) $\notin$ *set* (*arcs b a xs*) (*j*, *i*) $\notin$ *set* (*arcs b a xs*)
with ‹*i* $\neq$ *j*› **have** *len ?M' b a xs* = *len M b a xs* **by** (*induction xs arbitrary*: *b*) *auto*
} note $*$ = *this*
{ fix *a b xs* **assume** *A*: *i* $\notin$ *set* (*a* # *xs*) *j* $\notin$ *set* (*a* # *xs*)
then have *len ?M' a b xs* = *len M a b xs* **by** (*induction xs arbitrary*: *a*, *auto*)
} note $**$ = *this*
{ assume *A*:(*i*, *j*) $\notin$ *set* (*arcs a a xs*) (*j*, *i*) $\notin$ *set* (*arcs a a xs*)
from $*$[*OF this*] **have** *len ?M' a a xs* = *len M a a xs* **.**
with ‹*cycle-free M n*› *prems cycle* **have** *False* **by** (*auto simp*: *cycle-free-def*)
}
then have *arcs*:(*i*, *j*) $\in$ *set* (*arcs a a xs*) $\vee$ (*j*, *i*) $\in$ *set* (*arcs a a xs*) **by** *auto*
with ‹*i* $\neq$ *j*› **have** *xs* $\neq$ [] **by** *auto*
from *arcs* **obtain** *xs* **where** *xs*: *set xs* $\subseteq$ {*0..n*}
len ?M' i i (*j* # *xs*) < *0* $\vee$ *len ?M' j j* (*i* # *xs*) < *0*

proof (*standard*, *goal-cases*)
case *1*
from *cycle-rotate-2'*[*OF* ‹*xs* ≠ []› *this*(*2*), *of ?M'*] *prems* **obtain** *ys*
where
len ?M' i i (*j* # *ys*) = *len ?M' a a xs set ys* ⊆ {*0..n*}
by *fastforce*
with *1 cycle* **show** *?thesis* **by** *fastforce*
next
case *2*
from *cycle-rotate-2'*[*OF* ‹*xs* ≠ []› *this*(*2*), *of ?M'*] *prems* **obtain** *ys*
where
len ?M' j j (*i* # *ys*) = *len ?M' a a xs set ys* ⊆ {*0..n*}
by *fastforce*
with *2 cycle* **show** *?thesis* **by** *fastforce*
qed
from *new-negative-cycle-aux'*[*OF* ‹*i* ≤ *n*› ‹*j* ≤ *n*› *this*(*1*) ‹*cycle-free M n*› - *this*(*2*) ‹*i* ≠ *j*›]
obtain *xs* **where** *xs*:
set xs ⊆ {*0..n*} *i* ∉ *set xs j* ∉ *set xs*
len ?M' i i (*j* # *xs*) < *0* ∨ *len ?M' j j* (*i* # *xs*) < *0*
by *auto*
from *this*(*4*) **have** *False*
proof
assume *A*: *len ?M' j j* (*i* # *xs*) < *0*
show *False*
proof (*cases xs*)
case *Nil*
with ‹*i* ≠ *j*› **have** ∗:*?M' j i* = *Le* (−*r*) *?M' i j* = *Le r* **by** *simp*+
from *Nil* **have** *len ?M' j j* (*i* # *xs*) = *?M' j i* + *?M' i j* **by** *simp*
with ∗ **have** *len ?M' j j* (*i* # *xs*) = *Le 0* **by** (*simp add*: *add*)
then show *False* **using** *A* **by** (*simp add*: *neutral*)
next
case (*Cons y ys*)
have ∗:*M i y* + *M y j* ≥ *M i j*
using ‹*canonical M n*› *Cons xs* ‹*i* ≤ *n*› ‹*j* ≤ *n*› **by** (*simp add*: *add less-eq*)
have *Le 0* = *Le* (−*r*) + *Le r* **by** (*simp add*: *add*)
also have ... ≤ *Le* (−*r*) + *M i j* **using** *r* **by** (*simp add*: *add-mono*)
also have ... ≤ *Le* (−*r*) + *M i y* + *M y j* **using** ∗ **by** (*simp add*: *add-mono add.assoc*)
also have ... ≤ *Le* (−*r*) + *?M' i y* + *len M y j ys*
using *canonical-len*[*OF* ‹*canonical M n*›] *xs*(*1*−*3*) ‹*i* ≤ *n*› ‹*j* ≤ *n*› *Cons*
by (*simp add*: *add-mono*)

```
      also have ... = len ?M' j j (i # xs) using Cons ‹i ≠ j› ** xs(1−3)
        by (simp add: add.assoc)
      also have ... < Le 0 using A by (simp add: neutral)
      finally show False by simp
    qed
  next
    assume A: len ?M' i i (j # xs) < 0
    show False
    proof (cases xs)
      case Nil
      with ‹i ≠ j› have *:?M' j i = Le (−r) ?M' i j = Le r by simp+
      from Nil have len ?M' i i (j # xs) = ?M' i j + ?M' j i by simp
      with * have len ?M' i i (j # xs) = Le 0 by (simp add: add)
      then show False using A by (simp add: neutral)
    next
      case (Cons y ys)
      have *:M j y + M y i ≥ M j i
       using ‹canonical M n› Cons xs ‹i ≤ n› ‹j ≤ n› by (simp add: add
less-eq)
      have Le 0 = Le r + Le (−r) by (simp add: add)
      also have ... ≤ Le r + M j i using r by (simp add: add-mono)
       also have ... ≤ Le r + M j y + M y i using * by (simp add:
add-mono add.assoc)
      also have ... ≤ Le r + ?M' j y + len M y i ys
       using canonical-len[OF ‹canonical M n›] xs(1−3) ‹i ≤ n› ‹j ≤ n›
Cons
        by (simp add: add-mono)
      also have ... = len ?M' i i (j # xs) using Cons ‹i ≠ j› ** xs(1−3)
        by (simp add: add.assoc)
      also have ... < Le 0 using A by (simp add: neutral)
      finally show False by simp
    qed
  qed
 } note * = this
 have cycle-free ?M' n unfolding cycle-free-diag-equiv[symmetric]
   using negative-cycle-dest-diag * by fastforce
 then show ?thesis using not-empty ‹i ≠ j› r unfolding M'-def by auto
qed

lemma fix-index:
 fixes M :: (('a :: time) DBMEntry) mat
 assumes M 0 i + M i 0 > 0 cycle-free M n canonical M n i ≤ n i ≠ 0
 shows
 ∃ (M' :: ('a DBMEntry) mat). ((∃ u. DBM-val-bounded v u M' n) ⟶
```

$(\exists\ u.\ DBM\text{-}val\text{-}bounded\ v\ u\ M\ n))$
  $\wedge\ M'\ 0\ i + M'\ i\ 0 = 0 \wedge cycle\text{-}free\ M'\ n$
  $\wedge\ (\forall\ j.\ i \neq j \wedge M\ 0\ j + M\ j\ 0 = 0 \longrightarrow M'\ 0\ j + M'\ j\ 0 = 0)$
  $\wedge\ (\forall\ j.\ i \neq j \wedge M\ 0\ j + M\ j\ 0 > 0 \longrightarrow M'\ 0\ j + M'\ j\ 0 > 0)$
proof –
 note *A* = *assms*
 from *sum-gt-neutral-dest*[*OF assms*(*1*)] **obtain** *d* **where** *d*: *Le d* ≤ *M i 0 Le* (−*d*) ≤ *M 0 i* **by** *auto*
 have *i* ≠ *0* **using** *A* **by** – (*rule ccontr*; *simp*)
 let *?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′*
 from *fix-index′*[*OF d*(*1*,*2*) *A*(*2*,*3*,*4*) - ‹*i* ≠ *0*›] **have** *M′*:
  ∀ *u*. *DBM-val-bounded v u ?M′ n* ⟶ *DBM-val-bounded v u M n cycle-free ?M′ n*
 by *auto*
 moreover from ‹*i* ≠ *0*› **have** ∀ *j*. *i* ≠ *j* ∧ *M 0 j* + *M j 0* = *0* ⟶ *?M′ 0 j* + *?M′ j 0* = *0* **by** *auto*
 moreover from ‹*i* ≠ *0*› **have** ∀ *j*. *i* ≠ *j* ∧ *M 0 j* + *M j 0* > *0* ⟶ *?M′ 0 j* + *?M′ j 0* > *0* **by** *auto*
 moreover from ‹*i* ≠ *0*› **have** *?M′ 0 i* + *?M′ i 0* = *0* **unfolding** *neutral add* **by** *auto*
 ultimately show *?thesis* **by** *blast*
qed

Putting it together **lemma** *FW-not-empty*:
 DBM-val-bounded v u (*FW M′ n*) *n* ⟹ *DBM-val-bounded v u M′ n*
proof –
 assume *A*: *DBM-val-bounded v u* (*FW M′ n*) *n*
 have ∀ *i j*. *i* ≤ *n* ⟶ *j* ≤ *n* ⟶ *FW M′ n i j* ≤ *M′ i j* **using** *fw-mono* **by** *blast*
 from *DBM-le-subset*[*of n FW M′ n M′* - *v*, *OF this*[*unfolded less-eq*]]
 show *DBM-val-bounded v u M′ n* **using** *A* **by** (*auto simp*: *DBM-zone-repr-def*)
qed

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

using *assms*
proof (*induction xs arbitrary*: *M*)
  case *Nil* **then show** *?case* **by** *auto*
next
  case (*Cons i xs*)
  show *?case*
  proof (*cases M 0 i* + *M i 0* ≤ *0* ∨ *i* = *0*)
    case *True*
    note *T* = *this*
    show *?thesis*
    proof (*cases i* = *0*)
      case *False*
      from *Cons.prems* **have** *0* ≤ *n set* [*i*] ⊆ {*0..n*} **by** *auto*
      with *Cons.prems*(*3*) *False T* **have** *M 0 i* + *M i 0* = *0* **by** *fastforce*
     with *Cons.IH*[*OF - - Cons.prems*(*3*,*4*)] *Cons.prems*(*1*,*2*) **show** *?thesis*
by *auto*
    next
      case *True*
     with *Cons.IH*[*OF - - Cons.prems*(*3*,*4*)] *Cons.prems*(*1*,*2*) **show** *?thesis*
by *auto*
    qed
  next
    case *False*
    with *Cons.prems* **have** *0* < *M 0 i* + *M i 0 i* ≤ *n i* ≠ *0* **by** *auto*
   with *fix-index*[*OF this*(*1*) *cycle-free-diag-intro*[*OF Cons.prems*(*3*)] *Cons.prems*(*4*)
this(*2*,*3*), *of v*]
    obtain *M′* :: (*′a DBMEntry*) *mat* **where** *M′*:
      ((∃ *u*. *DBM-val-bounded v u M′ n*) ⟶ (∃ *u*. *DBM-val-bounded v u M*
n)) (*M′ 0 i* + *M′ i 0* = *0*)
      cyc-free M′ n ∀ *j*≤*n*. *i* ≠ *j* ∧ *M 0 j* + *M j 0* > *0* ⟶ *M′ 0 j* + *M′ j*
0 > *0*
     ∀ *j*. *i* ≠ *j* ∧ *M 0 j* + *M j 0* = *0* ⟶ *M′ 0 j* + *M′ j 0* = *0*
    using *cycle-free-diag-equiv* **by** *blast*
    let *?M′* = *FW M′ n*
    from *fw-canonical*[*of n M′*] ‹*cyc-free M′ n*› **have** *canonical ?M′ n* **by**
auto
    from *FW-cyc-free-preservation*[*OF* ‹*cyc-free M′ n*›] **have** *cyc-free ?M′*
n
    by *auto*
      from *FW-fixed-preservation*[*OF* ‹*i* ≤ *n*› *M′*(*2*) ‹*canonical ?M′ n*›
‹*cyc-free ?M′ n*›]
    have *fixed*:*?M′ 0 i* + *?M′ i 0* = *0* **by** (*auto simp*: *add-mono*)
   from *Cons.IH*[*OF - -* ‹*cyc-free ?M′ n*› ‹*canonical ?M′ n*›] *Cons.prems*(*1*,*2*,*3*)
    obtain *M″* :: (*′a DBMEntry*) *mat*

where M'': $((\exists u.\ DBM\text{-}val\text{-}bounded\ v\ u\ M''\ n) \longrightarrow (\exists u.\ DBM\text{-}val\text{-}bounded\ v\ u\ ?M'\ n))$
$(\forall i{\in}set\ xs.\ i \neq 0 \longrightarrow M''\ 0\ i + M''\ i\ 0 = 0)$ *cyc-free* $M''\ n$
$(\forall i{\le}n.\ i \notin set\ xs \wedge ?M'\ 0\ i + ?M'\ i\ 0 = 0 \longrightarrow M''\ 0\ i + M''\ i\ 0 = 0)$
by *auto*
from *FW-fixed-preservation*[*OF - - ‹canonical ?M′ n› ‹cyc-free ?M′ n›*] $M'(5)$
have $\forall j{\le}n.\ i \neq j \wedge M\ 0\ j + M\ j\ 0 = 0 \longrightarrow ?M'\ 0\ j + ?M'\ j\ 0 = 0$ **by** *auto*
with $M''(4)$ **have** $\forall j{\le}n.\ j \notin set\ (i\ \#\ xs) \wedge M\ 0\ j + M\ j\ 0 = 0 \longrightarrow M''\ 0\ j + M''\ j\ 0 = 0$ **by** *auto*
moreover from $M''(2)$ $M''(4)$ *fixed Cons.prems*(2) ‹$i \le n$›
have $(\forall i{\in}set\ (i\#xs).\ i \neq 0 \longrightarrow M''\ 0\ i + M''\ i\ 0 = 0)$ **by** *auto*
moreover from $M''(1)$ $M'(1)$ *FW-not-empty*[*of v - M′ n*]
have $(\exists u.\ DBM\text{-}val\text{-}bounded\ v\ u\ M''\ n) \longrightarrow (\exists u.\ DBM\text{-}val\text{-}bounded\ v\ u\ M\ n)$ **by** *auto*
ultimately show *?thesis* **using** ‹*cyc-free* M'' *n*› $M''(4)$ **by** *auto*
qed
qed

lemma *cyc-free-obtains-valuation*:
$cyc\text{-}free\ M\ n \Longrightarrow \forall\ c.\ v\ c \le n \longrightarrow v\ c > 0 \Longrightarrow \exists\ u.\ DBM\text{-}val\text{-}bounded\ v\ u\ M\ n$
proof –
assume *A*: $cyc\text{-}free\ M\ n\ \forall\ c.\ v\ c \le n \longrightarrow v\ c > 0$
let $?M = FW\ M\ n$
from *fw-canonical*[*of n M*] *A* **have** *canonical ?M n* **by** *auto*
from *FW-cyc-free-preservation*[*OF A*(1)] **have** *cyc-free ?M n* **.**
have $set\ [0..{<}n{+}1] \subseteq \{0..n\}$ $distinct\ [0..{<}n{+}1]$ **by** *auto*
from *fix-indices*[*OF this ‹cyc-free ?M n› ‹canonical ?M n›*]
obtain M' :: (*′a DBMEntry*) *mat* **where** M':
$(\exists u.\ DBM\text{-}val\text{-}bounded\ v\ u\ M'\ n) \longrightarrow (\exists u.\ DBM\text{-}val\text{-}bounded\ v\ u\ (FW\ M\ n)\ n)$
$\forall i{\in}set\ [0..{<}n + 1].\ i \neq 0 \longrightarrow M'\ 0\ i + M'\ i\ 0 = 0$ *cyc-free* $M'\ n$
by *blast*
let $?M' = FW\ M'\ n$
have $\bigwedge i.\ i \le n \Longrightarrow i \in set\ [0..{<}n + 1]$ **by** *auto*
with $M'(2)$ **have** M'-*fixed*: $\forall i{\le}n.\ i \neq 0 \longrightarrow M'\ 0\ i + M'\ i\ 0 = 0$ **by** *fastforce*
from *fw-canonical*[*of n M′*] $M'(3)$ **have** *canonical ?M′ n* **by** *blast*
from *FW-fixed-preservation*[*OF - - this FW-cyc-free-preservation*[*OF M′*(3)]] M'-*fixed*
have *fixed*: $\forall i{\le}n.\ i \neq 0 \longrightarrow ?M'\ 0\ i + ?M'\ i\ 0 = 0$ **by** *auto*

have ∗: $\bigwedge i.\ i \le n \Longrightarrow i \ne 0 \Longrightarrow \exists\ d.\ ?M'\ 0\ i = Le\ (-d) \wedge ?M'\ i\ 0 =$ $Le\ d$
proof −
fix *i* **assume** *i*: $i \le n\ i \ne 0$
from *i fixed* **have** ∗:*dbm-add* $(?M'\ 0\ i)\ (?M'\ i\ 0) = Le\ 0$ **by** (*auto simp add*: *add neutral*)
moreover
{ fix *a b* :: *'a* **assume** $a + b = 0$
then have $a = -b$ **by** (*simp add*: *eq-neg-iff-add-eq-0*)
}
ultimately show $\exists\, d.\ ?M'\ 0\ i = Le\ (-d) \wedge ?M'\ i\ 0 = Le\ d$
by (*cases ?M' 0 i*; *cases ?M' i 0*; *simp*)
qed
then obtain *f* **where** *f*: $\forall\ i{\le}n.\ i \ne 0 \longrightarrow Le\ (f\ i) = ?M'\ i\ 0 \wedge Le\ (-$ $f\ i) = ?M'\ 0\ i$ **by** *metis*
let $?u = \lambda\ c.\ f\ (v\ c)$
have *DBM-val-bounded v ?u ?M' n*
unfolding *DBM-val-bounded-def*
proof (*safe*, *goal-cases*)
case *1*
from *cyc-free-diag-dest'*[*OF FW-cyc-free-preservation*[*OF M'*(*3*)]] **show** *?case*
unfolding *neutral less-eq* **by** *fast*
next
case (*2 c*)
with *A*(*2*) **have** ∗∗: $v\ c > 0$ **by** *auto*
with ∗[*OF 2*] **obtain** *d* **where** *d*: $Le\ (-d) = ?M'\ 0\ (v\ c)$ **by** *auto*
with *f 2* ∗∗ **have** $Le\ (-\ f\ (v\ c)) = Le\ (-\ d)$ **by** *simp*
then have $-\ f\ (v\ c) \le -\ d$ **by** *auto*
from *dbm-entry-val.intros*(*2*)[*of ?u* , *OF this*] *d*
show *?case* **by** *auto*
next
case (*3 c*)
with *A*(*2*) **have** ∗∗: $v\ c > 0$ **by** *auto*
with ∗[*OF 3*] **obtain** *d* **where** *d*: $Le\ d = ?M'\ (v\ c)\ 0$ **by** *auto*
with *f 3* ∗∗ **have** $Le\ (f\ (v\ c)) = Le\ d$ **by** *simp*
then have $f\ (v\ c) \le d$ **by** *auto*
from *dbm-entry-val.intros*(*1*)[*of ?u*, *OF this*] *d*
show *?case* **by** *auto*
next
case (*4 c1 c2*)
with *A*(*2*) **have** ∗∗: $v\ c1 > 0\ v\ c2 > 0$ **by** *auto*
with ∗[*OF 4*(*1*)] **obtain** *d1* **where** *d1*: $Le\ d1 = ?M'\ (v\ c1)\ 0$ **by** *auto*
with *f 4* ∗∗ **have** $Le\ (f\ (v\ c1)) = Le\ d1$ **by** *simp*

then have *d1'*: *f* (*v c1*) = *d1* **by** *auto*
from ∗[*OF 4*(*2*)] ∗∗ **obtain** *d2* **where** *d2*: *Le d2* = *?M'* (*v c2*) *0* **by** *auto*
with *f 4* ∗∗ **have** *Le* (*f* (*v c2*)) = *Le d2* **by** *simp*
then have *d2'*: *f* (*v c2*) = *d2* **by** *auto*
have *Le d1* ≤ *?M'* (*v c1*) (*v c2*) + *Le d2* **using** ‹*canonical ?M' n*› *4 d1 d2*
by (*auto simp add*: *less-eq add*)
then show *?case*
proof (*cases ?M'* (*v c1*) (*v c2*), *goal-cases*)
case (*1 d*)
then have *d1* ≤ *d* + *d2* **by** (*auto simp*: *add less-eq le-dbm-le*)
then have *d1* − *d2* ≤ *d* **by** (*simp add*: *diff-le-eq*)
with *1* **show** *?case* **using** *d1' d2'* **by** *auto*
next
case (*2 d*)
then have *d1* < *d* + *d2* **by** (*auto simp*: *add less-eq dbm-le-def elim*: *dbm-lt.cases*)
then have *d1* − *d2* < *d* **using** *diff-less-eq* **by** *blast*
with *2* **show** *?case* **using** *d1' d2'* **by** *auto*
qed *auto*
qed
from *M'*(*1*) *FW-not-empty*[*OF this*] **obtain** *u* **where** *DBM-val-bounded v u ?M n* **by** *auto*
from *FW-not-empty*[*OF this*] **show** *?thesis* **by** *auto*
qed

2.6.1 Floyd-Warshall and Empty DBMs

theorem *FW-detects-empty-zone*:
∀ *k*≤*n*. *0* < *k* ⟶ (∃ *c*. *v c* = *k*) ⟹ ∀ *c*. *v c* ≤ *n* ⟶ *v c* > *0*
⟹ $[FW\ M\ n]_{v,n}$ = {} ⟷ (∃ *i*≤*n*. (*FW M n*) *i i* < *Le 0*)
proof
assume *surj-on*:∀ *k*≤*n*. *0* < *k* ⟶ (∃ *c*. *v c* = *k*) **and** ∃ *i*≤*n*. (*FW M n*) *i i* < *Le 0*
then obtain *i* **where** ∗: *len* (*FW M n*) *i i* [] < *0 i* ≤*n* **by** (*auto simp add*: *neutral*)
show $[FW\ M\ n]_{v,n}$ = {}
proof (*rule ccontr*, *goal-cases*)
case *1*
then obtain *u* **where** *DBM-val-bounded v u* (*FW M n*) *n* **unfolding** *DBM-zone-repr-def* **by** *auto*
from *DBM-val-bounded-neg-cycle*[*OF this* ∗(*2*) - ∗(*1*) *surj-on*] **show** *?case* **by** *auto*

qed
next
assume *surj-on*: ∀ *k*≤*n*. *0* < *k* ⟶ (∃ *c*. *v c* = *k*) **and** *empty*: $[FW\ M\ n]_{v,n}$ = {}
and *cn*: ∀ *c*. *v c* ≤ *n* ⟶ *v c* > *0*
show ∃ *i*≤*n*. (*FW M n*) *i i* < *Le 0*
proof (*rule ccontr*, *goal-cases*)
case *1*
then have ∗:∀ *i*≤*n*. *FW M n i i* ≥ *0* **by** (*auto simp add*: *neutral*)
have *cyc-free M n*
proof (*rule ccontr*)
assume ¬ *cyc-free M n*
from *FW-neg-cycle-detect*[*OF this*] ∗ **show** *False* **by** *auto*
qed
from *FW-cyc-free-preservation*[*OF this*] **have** *cyc-free* (*FW M n*) *n* **.**
from *cyc-free-obtains-valuation*[*OF* ‹*cyc-free* (*FW M n*) *n*› *cn*] *empty*
obtain *u* **where** *DBM-val-bounded v u* (*FW M n*) *n* **by** *blast*
with *empty* **show** *?case* **by** (*auto simp add*: *DBM-zone-repr-def*)
qed
qed

hide-const (**open**) *D*

2.6.2 Mixed Corollaries

lemma *cyc-free-not-empty*:
assumes *cyc-free M n* ∀ *c*. *v c* ≤ *n* ⟶ *0* < *v c*
shows $[(M :: ('a :: time)\ DBM)]_{v,n} \neq \{\}$
using *cyc-free-obtains-valuation*[*OF assms*(*1*,*2*)] **unfolding** *DBM-zone-repr-def*
by *auto*

lemma *empty-not-cyc-free*:
assumes ∀ *c*. *v c* ≤ *n* ⟶ *0* < *v c* $[(M :: ('a :: time)\ DBM)]_{v,n} = \{\}$
shows ¬ *cyc-free M n*
using *assms* **by** (*meson cyc-free-not-empty*)

lemma *not-empty-cyc-free*:
assumes ∀ *k*≤*n*. *0* < *k* ⟶ (∃ *c*. *v c* = *k*) $[(M :: ('a :: time)\ DBM)]_{v,n} \neq \{\}$
shows *cyc-free M n* **using** *DBM-val-bounded-neg-cycle*[*OF - - - - assms*(*1*)] *assms*(*2*)
unfolding *DBM-zone-repr-def* **by** *fastforce*

```
lemma neg-cycle-empty:
  assumes ∀k≤n. 0 < k ⟶ (∃ c. v c = k) set xs ⊆ {0..n} i ≤ n len M i
i xs < 0
  shows [(M :: ('a :: time) DBM)]_{v,n} = {} using assms
by (metis leD not-empty-cyc-free)

abbreviation clock-numbering' :: ('c ⇒ nat) ⇒ nat ⇒ bool
where
  clock-numbering' v n ≡ ∀ c. v c > 0 ∧ (∀x. ∀y. v x ≤ n ∧ v y ≤ n ∧ v
x = v y ⟶ x = y)

lemma non-empty-dbm-diag-set:
  clock-numbering' v n ⟹ [M]_{v,n} ≠ {}
  ⟹ [M]_{v,n} = [(λ i j. if i = j then 0 else M i j)]_{v,n}
  unfolding DBM-zone-repr-def
proof (safe, goal-cases)
  case 1
  { fix c assume A: v c = 0
    from 1 have v c > 0 by auto
    with A have False by auto
  } note * = this
  from 1 have [simp]: Le 0 ≼ M 0 0 by (auto simp: DBM-val-bounded-def)
  note [simp] = neutral
  from 1 show ?case
    unfolding DBM-val-bounded-def
    apply safe
    subgoal
      using * by simp
    subgoal
      using * by (metis (full-types))
    subgoal
      using * by (metis (full-types))
    subgoal for c1 c2
      by (cases c1 = c2) auto
    done
next
  case (2 x xa)
  note G = this
  { fix c assume A: v c = 0
    from 2 have v c > 0 by auto
    with A have False by auto
  } note * = this
  { fix c assume A: v c ≤ n M (v c) (v c) < 0
    with 2 have False
```

by (*fastforce simp*: *neutral DBM-val-bounded-def less elim*!: *dbm-lt.cases*)
} **note** $**$ = *this*
from *2* **have** [*simp*]: *Le 0* $\preceq$ *M 0 0* **by** (*auto simp*: *DBM-val-bounded-def*)
note [*simp*] = *neutral*
from *2* **show** *?case*
unfolding *DBM-val-bounded-def*
proof (*safe*, *goal-cases*)
case *1* **with** $*$ **show** *?case* **by** *simp presburger*
case *2* **with** $*$ **show** *?case* **by** *presburger*
next
case (*3 c1 c2*)
show *?case*
proof (*cases v c1 = v c2*)
case *True*
with *3* **have** *c1 = c2* **by** *auto*
moreover from *this* $**$[*OF 3*(*9*)] *not-less* **have** *M* (*v c2*) (*v c2*) $\geq$ *0* **by** *auto*
ultimately show *dbm-entry-val xa* (*Some c1*) (*Some c2*) (*M* (*v c1*) (*v c2*)) **unfolding** *neutral*
by (*cases M* (*v c1*) (*v c2*)) (*auto simp add*: *less-eq dbm-le-def*, *fastforce*+)
next
case *False*
with *3* **show** *?thesis* **by** *presburger*
qed
qed
qed

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*
apply (*rule ccontr*)
apply (*drule negative-cycle-dest-diag*′)
using *DBM-val-bounded-neg-cycle assms* **unfolding** *DBM-zone-repr-def* **by** *blast*

lemma *neg-diag-empty*:
assumes $\forall k \leq n.\ 0 < k \longrightarrow (\exists c.\ v\ c = k)$ $i \leq n$ $M\ i\ i < 0$
shows $[M]_{v,n} = \{\}$
unfolding *DBM-zone-repr-def* **using** *DBM-val-bounded-neg-cycle*[*of v - M n i* []] *assms* **by** *auto*

lemma *canonical-empty-zone*:

assumes $\forall k \le n.\ 0 < k \longrightarrow (\exists c.\ v\ c = k)$ $\forall c.\ v\ c \le n \longrightarrow 0 < v\ c$
and *canonical M n*
shows $[M]_{v,n} = \{\} \longleftrightarrow (\exists i \le n.\ M\ i\ i < 0)$
using *FW-detects-empty-zone*[*OF assms*(*1*,*2*), *of M*] *FW-canonical-id*[*OF assms*(*3*)] **unfolding** *neutral*
by *simp*

2.7 Orderings of DBMs

lemma *canonical-saturated-1*:
assumes $Le\ r \le M\ (v\ c1)\ 0$
and $Le\ (-\ r) \le M\ 0\ (v\ c1)$
and *cycle-free M n*
and *canonical M n*
and $v\ c1 \le n$
and $v\ c1 > 0$
and $\forall c.\ v\ c \le n \longrightarrow 0 < v\ c$
obtains *u* **where** $u \in [M]_{v,n}$ $u\ c1 = r$
proof −
let $?M' = \lambda i'\ j'.\ if\ i'=v\ c1 \wedge j'=0\ then\ Le\ r\ else\ if\ i'=0 \wedge j'=v\ c1\ then\ Le\ (-\ r)\ else\ M\ i'\ j'$
from *fix-index'*[*OF assms*(*1*−*5*)] *assms*(*6*) **have** *M'*:
$\forall u.$ *DBM-val-bounded v u ?M' n* $\longrightarrow$ *DBM-val-bounded v u M n*
cycle-free ?M' n $?M'\ (v\ c1)\ 0 = Le\ r$ $?M'\ 0\ (v\ c1) = Le\ (-\ r)$
by *auto*
with *cyc-free-obtains-valuation*[*unfolded cycle-free-diag-equiv*, *of ?M' n v*] *assms*(*7*) **obtain** *u* **where**
u: *DBM-val-bounded v u ?M' n*
by *fastforce*
with *assms*(*5*,*6*) *M'*(*3*,*4*) **have** $u\ c1 = r$ **unfolding** *DBM-val-bounded-def* **by** *fastforce*
moreover from *u M'*(*1*) **have** $u \in [M]_{v,n}$ **unfolding** *DBM-zone-repr-def* **by** *auto*
ultimately show *thesis* **by** (*auto intro*: *that*)
qed

lemma *canonical-saturated-2*:
assumes $Le\ r \le M\ 0\ (v\ c2)$
and $Le\ (-\ r) \le M\ (v\ c2)\ 0$
and *cycle-free M n*
and *canonical M n*
and $v\ c2 \le n$
and $v\ c2 > 0$
and $\forall c.\ v\ c \le n \longrightarrow 0 < v\ c$

obtains *u* **where** $u \in [M]_{v,n}$ *u c2* = − *r*
proof −
let *?M′* = λ*i′ j′*. *if i′=0* ∧ *j′=v c2 then Le r else if i′=v c2* ∧ *j′=0 then Le* (−*r*) *else M i′ j′*
from *fix-index′*[*OF assms*(*1*−*4*)] *assms*(*5*,*6*) **have** *M′*:
∀ *u*. *DBM-val-bounded v u ?M′ n* ⟶ *DBM-val-bounded v u M n*
cycle-free ?M′ n ?M′ 0 (*v c2*) = *Le r ?M′* (*v c2*) *0* = *Le* (− *r*)
by *auto*
with *cyc-free-obtains-valuation*[*unfolded cycle-free-diag-equiv*, *of ?M′ n v*] *assms*(*7*) **obtain** *u* **where**
u: *DBM-val-bounded v u ?M′ n*
by *fastforce*
with *assms*(*5*,*6*) *M′*(*3*,*4*) **have** *u c2* ≤ −*r* − *u c2* ≤ *r* **unfolding** *DBM-val-bounded-def* **by** *fastforce*+
then have *u c2* = −*r* **by** (*simp add*: *le-minus-iff*)
moreover from *u M′*(*1*) **have** $u \in [M]_{v,n}$ **unfolding** *DBM-zone-repr-def* **by** *auto*
ultimately show *thesis* **by** (*auto intro*: *that*)
qed

lemma *canonical-saturated-3*:
assumes *Le r* ≤ *M* (*v c1*) (*v c2*)
and *Le* (− *r*) ≤ *M* (*v c2*) (*v c1*)
and *cycle-free M n*
and *canonical M n*
and *v c1* ≤ *n v c2* ≤ *n*
and *v c1* ≠ *v c2*
and ∀ *c*. *v c* ≤ *n* ⟶ *0* < *v c*
obtains *u* **where** $u \in [M]_{v,n}$ *u c1* − *u c2* = *r*
proof −
let *?M′*=λ*i′ j′*. *if i′=v c1* ∧ *j′=v c2 then Le r else if i′=v c2* ∧ *j′=v c1 then Le* (−*r*) *else M i′ j′*
from *fix-index′*[*OF assms*(*1*−*7*), *of v*] *assms*(*7*,*8*) **have** *M′*:
∀ *u*. *DBM-val-bounded v u ?M′ n* ⟶ *DBM-val-bounded v u M n*
cycle-free ?M′ n ?M′ (*v c1*) (*v c2*) = *Le r ?M′* (*v c2*) (*v c1*) = *Le* (−*r*)
by *auto*
with *cyc-free-obtains-valuation*[*unfolded cycle-free-diag-equiv*, *of ?M′ n v*] *assms* **obtain** *u* **where** *u*:
DBM-val-bounded v u ?M′ n
by *fastforce*
with *assms*(*5*,*6*) *M′*(*3*,*4*) **have**
u c1 − *u c2* ≤ *r u c2* − *u c1* ≤ − *r*
unfolding *DBM-val-bounded-def* **by** *fastforce*+

then have *u c1* − *u c2* = *r* **by** (*simp add*: *le-minus-iff*)
moreover from *u M′(1)* **have** $u \in [M]_{v,n}$ **unfolding** *DBM-zone-repr-def* **by** *auto*
ultimately show *thesis* **by** (*auto intro*: *that*)
qed

lemma *DBM-canonical-subset-le*:
notes *any-le-inf*[*intro*]
fixes *M* :: *real DBM*
assumes *canonical M n* $[M]_{v,n} \subseteq [M']_{v,n}$ $[M]_{v,n} \neq \{\}$ $i \leq n$ $j \leq n$ $i \neq j$
assumes *clock-numbering*: *clock-numbering′ v n*
$\forall k \leq n.\ 0 < k \longrightarrow (\exists c.\ v\ c = k)$
shows $M\ i\ j \leq M'\ i\ j$
proof −
from *non-empty-cycle-free*[*OF assms*(*3*)] *clock-numbering*(*2*) **have** *cycle-free M n* **by** *auto*
with *assms*(*1*,*4*,*5*) **have** *non-neg*:
$M\ i\ j + M\ j\ i \geq Le\ 0$
by (*metis cycle-free-diag order.trans neutral*)

from *clock-numbering* **have** *cn*: $\forall c.\ v\ c \leq n \longrightarrow 0 < v\ c$ **by** *auto*
show *?thesis*
proof (*cases i* = *0*)
case *True*
show *?thesis*
proof (*cases j* = *0*)
case *True*
with *assms* ‹*i* = *0*› **show** *?thesis*
unfolding *neutral DBM-zone-repr-def DBM-val-bounded-def less-eq* **by** *auto*
next
case *False*
then have *j* > *0* **by** *auto*
with ‹$j \leq n$› *clock-numbering* **obtain** *c2* **where** *c2*: *v c2* = *j* **by** *auto*
note *t* = *canonical-saturated-2*[*OF - -* ‹*cycle-free M n*› *assms*(*1*) *assms*(*5*)[*folded c2*] *- cn*,*unfolded c2*]
show *?thesis*
proof (*rule ccontr*, *goal-cases*)
case *1*
{ fix *d* **assume** *1*: $M\ 0\ j = \infty$
obtain *r* **where** *r*: $Le\ r \leq M\ 0\ j$ $Le\ (-r) \leq M\ j\ 0$ $d < r$
proof (*cases M j 0*)
case (*Le d′*)
obtain *r* **where** $r > -d'$ **using** *gt-ex* **by** *blast*

with *Le 1* **show** *?thesis* **by** (*intro that*[*of max r* (*d* + *1*)]) *auto*
next
case (*Lt* d')
obtain *r* **where** $r > - d'$ **using** *gt-ex* **by** *blast*
with *Lt 1* **show** *?thesis* **by** (*intro that*[*of max r* (*d* + *1*)]) *auto*
next
case *INF*
with *1* **show** *?thesis* **by** (*intro that*[*of d* + *1*]) *auto*
qed
then have $\exists\ r.\ Le\ r \leq M\ 0\ j \wedge Le\ (-r) \leq M\ j\ 0 \wedge d < r$ **by** *auto*
} **note** *inf-case* = *this*
{ **fix** *a b d* :: *real* **assume** *1*: $a < b$ **assume** *b*: $b + d > 0$
then have $*$: $b > -d$ **by** *auto*
obtain *r* **where** $r > - d$ $r > a$ $r < b$
proof (*cases* $a \geq - d$)
case *True*
from *1* **obtain** *r* **where** $r > a$ $r < b$ **using** *dense* **by** *auto*
with *True* **show** *?thesis* **by** (*auto intro*: *that*[*of r*])
next
case *False*
with $*$ **obtain** *r* **where** $r > -d$ $r < b$ **using** *dense* **by** *auto*
with *False* **show** *?thesis* **by** (*auto intro*: *that*[*of r*])
qed
then have $\exists\ r.\ r > -d \wedge r > a \wedge r < b$ **by** *auto*
} **note** *gt-case* = *this*
{ **fix** *a r* **assume** *r*: $Le\ r \leq M\ 0\ j$ $Le\ (-r) \leq M\ j\ 0$ $a < r$ $M'\ 0\ j = Le\ a \vee M'\ 0\ j = Lt\ a$
from *t*[*OF this*(*1*,*2*) ‹$0 < j$›] **obtain** *u* **where** *u*: $u \in [M]_{v,n}$ $u\ c2 = - r$.
with ‹$j \leq n$› *c2 assms*(*2*) **have** *dbm-entry-val u None* (*Some c2*) ($M'\ 0\ j$)
unfolding *DBM-zone-repr-def DBM-val-bounded-def* **by** *blast*
with *u*(*2*) *r*(*3*,*4*) **have** *False* **by** *auto*
} **note** *contr* = *this*
from *1 True* **have** $M'\ 0\ j < M\ 0\ j$ **by** *auto*
then show *False* **unfolding** *less*
proof (*cases rule*: *dbm-lt.cases*)
case (*1 d*)
with *inf-case* **obtain** *r* **where** *r*: $Le\ r \leq M\ 0\ j$ $Le\ (-r) \leq M\ j\ 0$ $d < r$ **by** *auto*
from *contr*[*OF this*] *1* **show** *False* **by** *fast*
next
case (*2 d*)
with *inf-case* **obtain** *r* **where** *r*: $Le\ r \leq M\ 0\ j$ $Le\ (-r) \leq M\ j\ 0$ *d*

$<$ *r* **by** *auto*
from *contr*[*OF this*] *2* **show** *False* **by** *fast*
next
case (*3 a b*)
obtain *r* **where** *r*: *Le r* $\leq$ *M 0 j Le* $(-r) \leq$ *M j 0 a* $<$ *r*
proof (*cases M j 0*)
case (*Le d'*)
with *3 non-neg* ‹$i = 0$› **have** $b + d' \geq 0$ **unfolding** *add* **by** *auto*
then have $b \geq - d'$ **by** *auto*
with *3* **obtain** *r* **where** $r \geq - d'$ $r > a$ $r \leq b$ **by** *blast*
with *Le 3* **show** *?thesis* **by** (*intro that*[*of r*]) *auto*
next
case (*Lt d'*)
with *3 non-neg* ‹$i = 0$› **have** $b + d' > 0$ **unfolding** *add* **by** *auto*
from *gt-case*[*OF 3*(*3*) *this*] **obtain** *r* **where** $r > - d'$ $r > a$ $r \leq$
b **by** *auto*
with *Lt 3* **show** *?thesis* **by** (*intro that*[*of r*]) *auto*
next
case *INF*
with *3* **show** *?thesis* **by** (*intro that*[*of b*]) *auto*
qed
from *contr*[*OF this*] *3* **show** *False* **by** *fast*
next
case (*4 a b*)
obtain *r* **where** *r*: *Le r* $\leq$ *M 0 j Le* $(-r) \leq$ *M j 0 a* $<$ *r*
proof (*cases M j 0*)
case (*Le d*)
with *4 non-neg* ‹$i = 0$› **have** $b + d > 0$ **unfolding** *add* **by** *auto*
from *gt-case*[*OF 4*(*3*) *this*] **obtain** *r* **where** $r > - d$ $r > a$ $r <$
b **by** *auto*
with *Le 4* **show** *?thesis* **by** (*intro that*[*of r*]) *auto*
next
case (*Lt d*)
with *4 non-neg* ‹$i = 0$› **have** $b + d > 0$ **unfolding** *add* **by** *auto*
from *gt-case*[*OF 4*(*3*) *this*] **obtain** *r* **where** $r > - d$ $r > a$ $r <$
b **by** *auto*
with *Lt 4* **show** *?thesis* **by** (*intro that*[*of r*]) *auto*
next
case *INF*
from *4 dense* **obtain** *r* **where** $r > a$ $r < b$ **by** *auto*
with *4 INF* **show** *?thesis* **by** (*intro that*[*of r*]) *auto*
qed
from *contr*[*OF this*] *4* **show** *False* **by** *fast*
next

```
case (5 a b)
obtain r where r: Le r ≤ M 0 j Le (−r) ≤ M j 0 a ≤ r
proof (cases M j 0)
  case (Le d′)
 with 5 non-neg ‹i = 0› have b + d′ ≥ 0 unfolding add by auto
  then have b ≥ − d′ by auto
  with 5 obtain r where r ≥ − d′ r ≥ a r ≤ b by blast
  with Le 5 show ?thesis by (intro that[of r]) auto
next
  case (Lt d′)
 with 5 non-neg ‹i = 0› have b + d′ > 0 unfolding add by auto
  then have b > − d′ by auto
  with 5 obtain r where r > − d′ r ≥ a r ≤ b by blast
  with Lt 5 show ?thesis by (intro that[of r]) auto
next
  case INF
  with 5 show ?thesis by (intro that[of b]) auto
qed
from t[OF this(1,2) ‹j > 0›] obtain u where u: u ∈ [M]v,n u c2
= − r .
 with ‹j ≤ n› c2 assms(2) have dbm-entry-val u None (Some c2)
(M′ 0 j)
unfolding DBM-zone-repr-def DBM-val-bounded-def by blast
with u(2) r(3) 5 show False by auto
next
case (6 a b)
obtain r where r: Le r ≤ M 0 j Le (−r) ≤ M j 0 a < r
proof (cases M j 0)
  case (Le d)
  with 6 non-neg ‹i = 0› have b + d > 0 unfolding add by auto
  from gt-case[OF 6(3) this] obtain r where r > − d r > a r <
b by auto
  with Le 6 show ?thesis by (intro that[of r]) auto
next
  case (Lt d)
  with 6 non-neg ‹i = 0› have b + d > 0 unfolding add by auto
  from gt-case[OF 6(3) this] obtain r where r > − d r > a r <
b by auto
  with Lt 6 show ?thesis by (intro that[of r]) auto
next
  case INF
  from 6 dense obtain r where r > a r < b by auto
  with 6 INF show ?thesis by (intro that[of r]) auto
qed
```

```
        from contr[OF this] 6 show False by fast
      qed
    qed
  qed
next
  case False
  then have i > 0 by auto
  with ‹i ≤ n› clock-numbering obtain c1 where c1: v c1 = i by auto
  show ?thesis
  proof (cases j = 0)
    case True
      note t = canonical-saturated-1[OF - - ‹cycle-free M n› assms(1)
assms(4)[folded c1] - cn,
                                      unfolded c1]
    show ?thesis
    proof (rule ccontr, goal-cases)
      case 1
      { fix d assume 1: M i 0 = ∞
        obtain r where r: Le r ≤ M i 0 Le (−r) ≤ M 0 i d < r
        proof (cases M 0 i)
          case (Le d')
          obtain r where r > − d' using gt-ex by blast
          with Le 1 show ?thesis by (intro that[of max r (d + 1)]) auto
        next
          case (Lt d')
          obtain r where r > − d' using gt-ex by blast
          with Lt 1 show ?thesis by (intro that[of max r (d + 1)]) auto
        next
          case INF
          with 1 show ?thesis by (intro that[of d + 1]) auto
        qed
       then have ∃ r. Le r ≤ M i 0 ∧ Le (−r) ≤ M 0 i ∧ d < r by auto
      } note inf-case = this
      { fix a b d :: real assume 1: a < b assume b: b + d > 0
        then have *: b > −d by auto
        obtain r where r > − d r > a r < b
        proof (cases a ≥ − d)
          case True
          from 1 obtain r where r > a r < b using dense by auto
          with True show ?thesis by (auto intro: that[of r])
        next
          case False
          with * obtain r where r > −d r < b using dense by auto
          with False show ?thesis by (auto intro: that[of r])
```

qed
then have $\exists\ r.\ r > -d \wedge r > a \wedge r < b$ **by** *auto*
} **note** *gt-case* = *this*
{ **fix** *a r* **assume** *r*: *Le r* $\leq$ *M i 0 Le* $(-r) \leq$ *M 0 i a* $<$ *r M′ i 0* = *Le a* $\vee$ *M′ i 0* = *Lt a*
from *t*[*OF this*(*1*,*2*) ‹*i* > *0*›] **obtain** *u* **where** *u*: $u \in [M]_{v,n}$ *u c1* = *r* **.**
with ‹*i* $\leq$ *n*› *c1 assms*(*2*) **have** *dbm-entry-val u* (*Some c1*) *None* (*M′ i 0*)
unfolding *DBM-zone-repr-def DBM-val-bounded-def* **by** *blast*
with *u*(*2*) *r*(*3*,*4*) **have** *False* **by** *auto*
} **note** *contr* = *this*
from *1 True* **have** *M′ i 0* $<$ *M i 0* **by** *auto*
then show *False* **unfolding** *less*
proof (*cases rule*: *dbm-lt.cases*)
case (*1 d*)
with *inf-case* **obtain** *r* **where** *r*: *Le r* $\leq$ *M i 0 Le* $(-r) \leq$ *M 0 i d* $<$ *r* **by** *auto*
from *contr*[*OF this*] *1* **show** *False* **by** *fast*
next
case (*2 d*)
with *inf-case* **obtain** *r* **where** *r*: *Le r* $\leq$ *M i 0 Le* $(-r) \leq$ *M 0 i d* $<$ *r* **by** *auto*
from *contr*[*OF this*] *2* **show** *False* **by** *fast*
next
case (*3 a b*)
obtain *r* **where** *r*: *Le r* $\leq$ *M i 0 Le* $(-r) \leq$ *M 0 i a* $<$ *r*
proof (*cases M 0 i*)
case (*Le d′*)
with *3 non-neg* ‹*j* = *0*› **have** $b + d' \geq 0$ **unfolding** *add* **by** *auto*
then have $b \geq -\ d'$ **by** *auto*
with *3* **obtain** *r* **where** $r \geq -\ d'$ $r > a$ $r \leq b$ **by** *blast*
with *Le 3* **show** *?thesis* **by** (*intro that*[*of r*]) *auto*
next
case (*Lt d′*)
with *3 non-neg* ‹*j* = *0*› **have** $b + d' > 0$ **unfolding** *add* **by** *auto*
from *gt-case*[*OF 3*(*3*) *this*] **obtain** *r* **where** $r > -\ d'$ $r > a$ $r \leq b$ **by** *auto*
with *Lt 3* **show** *?thesis* **by** (*intro that*[*of r*]) *auto*
next
case *INF*
with *3* **show** *?thesis* **by** (*intro that*[*of b*]) *auto*
qed
from *contr*[*OF this*] *3* **show** *False* **by** *fast*

next
case (*4 a b*)
obtain *r* **where** *r*: *Le r* ≤ *M i 0 Le* (−*r*) ≤ *M 0 i a* < *r*
proof (*cases M 0 i*)
case (*Le d*)
with *4 non-neg* ‹*j* = *0*› **have** *b* + *d* > *0* **unfolding** *add* **by** *auto*
from *gt-case*[*OF 4*(*3*) *this*] **obtain** *r* **where** *r* > − *d r* > *a r* < *b* **by** *auto*
with *Le 4* **show** *?thesis* **by** (*intro that*[*of r*]) *auto*
next
case (*Lt d*)
with *4 non-neg* ‹*j* = *0*› **have** *b* + *d* > *0* **unfolding** *add* **by** *auto*
from *gt-case*[*OF 4*(*3*) *this*] **obtain** *r* **where** *r* > − *d r* > *a r* < *b* **by** *auto*
with *Lt 4* **show** *?thesis* **by** (*intro that*[*of r*]) *auto*
next
case *INF*
from *4 dense* **obtain** *r* **where** *r* > *a r* < *b* **by** *auto*
with *4 INF* **show** *?thesis* **by** (*intro that*[*of r*]) *auto*
qed
from *contr*[*OF this*] *4* **show** *False* **by** *fast*
next
case (*5 a b*)
obtain *r* **where** *r*: *Le r* ≤ *M i 0 Le* (−*r*) ≤ *M 0 i a* ≤ *r*
proof (*cases M 0 i*)
case (*Le d′*)
with *5 non-neg* ‹*j* = *0*› **have** *b* + *d′* ≥ *0* **unfolding** *add* **by** *auto*
then have *b* ≥ − *d′* **by** *auto*
with *5* **obtain** *r* **where** *r* ≥ − *d′ r* ≥ *a r* ≤ *b* **by** *blast*
with *Le 5* **show** *?thesis* **by** (*intro that*[*of r*]) *auto*
next
case (*Lt d′*)
with *5 non-neg* ‹*j* = *0*› **have** *b* + *d′* > *0* **unfolding** *add* **by** *auto*
then have *b* > − *d′* **by** *auto*
with *5* **obtain** *r* **where** *r* > − *d′ r* ≥ *a r* ≤ *b* **by** *blast*
with *Lt 5* **show** *?thesis* **by** (*intro that*[*of r*]) *auto*
next
case *INF*
with *5* **show** *?thesis* **by** (*intro that*[*of b*]) *auto*
qed
from *t*[*OF this*(*1*,*2*) ‹*i* > *0*›] **obtain** *u* **where** *u*: *u* ∈ $[M]_{v,n}$ *u c1* = *r* **.**
with ‹*i* ≤ *n*› *c1 assms*(*2*) **have** *dbm-entry-val u* (*Some c1*) *None* (*M′ i 0*)

unfolding *DBM-zone-repr-def DBM-val-bounded-def* **by** *blast*
with *u*(*2*) *r*(*3*) *5* **show** *False* **by** *auto*
next
case (*6 a b*)
obtain *r* **where** *r*: *Le r* ≤ *M i 0 Le* (−*r*) ≤ *M 0 i a* < *r*
proof (*cases M 0 i*)
case (*Le d*)
with *6 non-neg* ‹*j* = *0*› **have** *b* + *d* > *0* **unfolding** *add* **by** *auto*
from *gt-case*[*OF 6*(*3*) *this*] **obtain** *r* **where** *r* > − *d r* > *a r* < *b* **by** *auto*
with *Le 6* **show** *?thesis* **by** (*intro that*[*of r*]) *auto*
next
case (*Lt d*)
with *6 non-neg* ‹*j* = *0*› **have** *b* + *d* > *0* **unfolding** *add* **by** *auto*
from *gt-case*[*OF 6*(*3*) *this*] **obtain** *r* **where** *r* > − *d r* > *a r* < *b* **by** *auto*
with *Lt 6* **show** *?thesis* **by** (*intro that*[*of r*]) *auto*
next
case *INF*
from *6 dense* **obtain** *r* **where** *r* > *a r* < *b* **by** *auto*
with *6 INF* **show** *?thesis* **by** (*intro that*[*of r*]) *auto*
qed
from *contr*[*OF this*] *6* **show** *False* **by** *fast*
qed
qed
next
case *False*
then have *j* > *0* **by** *auto*
with ‹*j* ≤ *n*› *clock-numbering* **obtain** *c2* **where** *c2*: *v c2* = *j* **by** *auto*
note *t* = *canonical-saturated-3*[*OF - -* ‹*cycle-free M n*› *assms*(*1*) *assms*(*4*)[*folded c1*]
assms(*5*)[*folded c2*] *- cn*, *unfolded c1 c2*]
show *?thesis*
proof (*rule ccontr*, *goal-cases*)
case *1*
{ fix *d* **assume** *1*: *M i j* = ∞
obtain *r* **where** *r*: *Le r* ≤ *M i j Le* (−*r*) ≤ *M j i d* < *r*
proof (*cases M j i*)
case (*Le d′*)
obtain *r* **where** *r* > − *d′* **using** *gt-ex* **by** *blast*
with *Le 1* **show** *?thesis* **by** (*intro that*[*of max r* (*d* + *1*)]) *auto*
next
case (*Lt d′*)
obtain *r* **where** *r* > − *d′* **using** *gt-ex* **by** *blast*

with *Lt 1* **show** *?thesis* **by** (*intro that*[*of max r* (*d* + *1*)]) *auto*
next
case *INF*
with *1* **show** *?thesis* **by** (*intro that*[*of d* + *1*]) *auto*
qed
then have ∃ *r*. *Le r* ≤ *M i j* ∧ *Le* (−*r*) ≤ *M j i* ∧ *d* < *r* **by** *auto*
} **note** *inf-case* = *this*
{ **fix** *a b d* :: *real* **assume** *1*: *a* < *b* **assume** *b*: *b* + *d* > *0*
then have ∗: *b* > −*d* **by** *auto*
obtain *r* **where** *r* > − *d r* > *a r* < *b*
proof (*cases a* ≥ − *d*)
case *True*
from *1* **obtain** *r* **where** *r* > *a r* < *b* **using** *dense* **by** *auto*
with *True* **show** *?thesis* **by** (*auto intro*: *that*[*of r*])
next
case *False*
with ∗ **obtain** *r* **where** *r* > −*d r* < *b* **using** *dense* **by** *auto*
with *False* **show** *?thesis* **by** (*auto intro*: *that*[*of r*])
qed
then have ∃ *r*. *r* > −*d* ∧ *r* > *a* ∧ *r* < *b* **by** *auto*
} **note** *gt-case* = *this*
{ **fix** *a r* **assume** *r*: *Le r* ≤ *M i j Le* (−*r*) ≤ *M j i a* < *r M′ i j* = *Le a* ∨ *M′ i j* = *Lt a*
from *t*[*OF this*(*1*,*2*) ‹*i* ≠ *j*›] **obtain** *u* **where** *u*: $u \in [M]_{v,n}$ *u c1* − *u c2* = *r* **.**
with ‹*i* ≤ *n*› ‹*j* ≤ *n*› *c1 c2 assms*(*2*) **have** *dbm-entry-val u* (*Some c1*) (*Some c2*) (*M′ i j*)
unfolding *DBM-zone-repr-def DBM-val-bounded-def* **by** *blast*
with *u*(*2*) *r*(*3*,*4*) **have** *False* **by** *auto*
} **note** *contr* = *this*
from *1* **have** *M′ i j* < *M i j* **by** *auto*
then show *False* **unfolding** *less*
proof (*cases rule*: *dbm-lt.cases*)
case (*1 d*)
with *inf-case* **obtain** *r* **where** *r*: *Le r* ≤ *M i j Le* (−*r*) ≤ *M j i d* < *r* **by** *auto*
from *contr*[*OF this*] *1* **show** *False* **by** *fast*
next
case (*2 d*)
with *inf-case* **obtain** *r* **where** *r*: *Le r* ≤ *M i j Le* (−*r*) ≤ *M j i d* < *r* **by** *auto*
from *contr*[*OF this*] *2* **show** *False* **by** *fast*
next
case (*3 a b*)

obtain *r* **where** *r*: *Le r* ≤ *M i j Le* (−*r*) ≤ *M j i a* < *r*
proof (*cases M j i*)
  case (*Le d′*)
  with *3 non-neg* **have** *b* + *d′* ≥ *0* **unfolding** *add* **by** *auto*
  then have *b* ≥ − *d′* **by** *auto*
  with *3* **obtain** *r* **where** *r* ≥ − *d′ r* > *a r* ≤ *b* **by** *blast*
  with *Le 3* **show** *?thesis* **by** (*intro that*[*of r*]) *auto*
next
  case (*Lt d′*)
  with *3 non-neg* **have** *b* + *d′* > *0* **unfolding** *add* **by** *auto*
  from *gt-case*[*OF 3*(*3*) *this*] **obtain** *r* **where** *r* > − *d′ r* > *a r* ≤
b **by** *auto*
  with *Lt 3* **show** *?thesis* **by** (*intro that*[*of r*]) *auto*
next
  case *INF*
  with *3* **show** *?thesis* **by** (*intro that*[*of b*]) *auto*
qed
from *contr*[*OF this*] *3* **show** *False* **by** *fast*
next
case (*4 a b*)
obtain *r* **where** *r*: *Le r* ≤ *M i j Le* (−*r*) ≤ *M j i a* < *r*
proof (*cases M j i*)
  case (*Le d*)
  with *4 non-neg* **have** *b* + *d* > *0* **unfolding** *add* **by** *auto*
  from *gt-case*[*OF 4*(*3*) *this*] **obtain** *r* **where** *r* > − *d r* > *a r* <
b **by** *auto*
  with *Le 4* **show** *?thesis* **by** (*intro that*[*of r*]) *auto*
next
  case (*Lt d*)
  with *4 non-neg* **have** *b* + *d* > *0* **unfolding** *add* **by** *auto*
  from *gt-case*[*OF 4*(*3*) *this*] **obtain** *r* **where** *r* > − *d r* > *a r* <
b **by** *auto*
  with *Lt 4* **show** *?thesis* **by** (*intro that*[*of r*]) *auto*
next
  case *INF*
  from *4 dense* **obtain** *r* **where** *r* > *a r* < *b* **by** *auto*
  with *4 INF* **show** *?thesis* **by** (*intro that*[*of r*]) *auto*
qed
from *contr*[*OF this*] *4* **show** *False* **by** *fast*
next
case (*5 a b*)
obtain *r* **where** *r*: *Le r* ≤ *M i j Le* (−*r*) ≤ *M j i a* ≤ *r*
proof (*cases M j i*)
  case (*Le d′*)

with *5 non-neg* **have** $b + d' \geq 0$ **unfolding** *add* **by** *auto*
then have $b \geq - d'$ **by** *auto*
with *5* **obtain** *r* **where** $r \geq - d'$ $r \geq a$ $r \leq b$ **by** *blast*
with *Le 5* **show** *?thesis* **by** (*intro that*[*of r*]) *auto*
next
case (*Lt d'*)
with *5 non-neg* **have** $b + d' > 0$ **unfolding** *add* **by** *auto*
then have $b > - d'$ **by** *auto*
with *5* **obtain** *r* **where** $r > - d'$ $r \geq a$ $r \leq b$ **by** *blast*
with *Lt 5* **show** *?thesis* **by** (*intro that*[*of r*]) *auto*
next
case *INF*
with *5* **show** *?thesis* **by** (*intro that*[*of b*]) *auto*
qed
from *t*[*OF this*(*1*,*2*) ‹$i \neq j$›] **obtain** *u* **where** *u*: $u \in [M]_{v,n}$ *u c1* − *u c2*= *r* **.**
with ‹$i \leq n$› ‹$j \leq n$› *c1 c2 assms*(*2*) **have** *dbm-entry-val u* (*Some c1*) (*Some c2*) (*M' i j*)
unfolding *DBM-zone-repr-def DBM-val-bounded-def* **by** *blast*
with *u*(*2*) *r*(*3*) *5* **show** *False* **by** *auto*
next
case (*6 a b*)
obtain *r* **where** *r*: *Le r* ≤ *M i j Le* (−*r*) ≤ *M j i a* < *r*
proof (*cases M j i*)
case (*Le d*)
with *6 non-neg* **have** $b + d > 0$ **unfolding** *add* **by** *auto*
from *gt-case*[*OF 6*(*3*) *this*] **obtain** *r* **where** $r > - d$ $r > a$ $r < b$ **by** *auto*
with *Le 6* **show** *?thesis* **by** (*intro that*[*of r*]) *auto*
next
case (*Lt d*)
with *6 non-neg* **have** $b + d > 0$ **unfolding** *add* **by** *auto*
from *gt-case*[*OF 6*(*3*) *this*] **obtain** *r* **where** $r > - d$ $r > a$ $r < b$ **by** *auto*
with *Lt 6* **show** *?thesis* **by** (*intro that*[*of r*]) *auto*
next
case *INF*
from *6 dense* **obtain** *r* **where** $r > a$ $r < b$ **by** *auto*
with *6 INF* **show** *?thesis* **by** (*intro that*[*of r*]) *auto*
qed
from *contr*[*OF this*] *6* **show** *False* **by** *fast*
qed
qed
qed

qed
qed

end
theory *FW-More*
imports
DBM-Basics
Floyd-Warshall.FW-Code
begin

2.8 Partial Floyd-Warshall Preserves Zones

lemma *fwi-len-distinct*:
$\exists$ *ys. set ys* $\subseteq \{k\} \land$ *fwi m n k n n i j* = *len m i j ys* $\land$ *i* $\notin$ *set ys* $\land$ *j* $\notin$ *set ys* $\land$ *distinct ys*
if $i \leq n$ $j \leq n$ $k \leq n$ *m k k* $\geq$ *0*
using *fwi-step'*[*of m*, *OF that*(*4*), *of n n n i j*] *that*
apply (*clarsimp split*: *if-splits simp*: *min-def*)
by (*rule exI*[**where** *x* = []] *exI*[**where** *x* = [*k*]]; *auto simp*: *add-increasing add-increasing2*)+

lemma *FWI-mono*:
$i \leq n \Longrightarrow j \leq n \Longrightarrow$ *FWI M n k i j* $\leq$ *M i j*
using *fwi-mono*[*of - n - M k n n*, *folded FWI-def*, *rule-format*] **.**

lemma *FWI-zone-equiv*:
$[M]_{v,n}$ = $[FWI\ M\ n\ k]_{v,n}$ **if** *surj-on*: $\forall$ $k \leq n$. $k > 0 \longrightarrow (\exists\ c.\ v\ c = k)$ **and** $k \leq n$
proof *safe*
fix *u* **assume** *A*: $u \in [FWI\ M\ n\ k]_{v,n}$
{ fix *i j* **assume** $i \leq n$ $j \leq n$
then have *FWI M n k i j* $\leq$ *M i j* **by** (*rule FWI-mono*)
hence *FWI M n k i j* $\preceq$ *M i j* **by** (*simp add*: *less-eq*)
}
with *DBM-le-subset*[*of n FWI M n k M*] *A* **show** $u \in [M]_{v,n}$ **by** *auto*
next
fix *u* **assume** *u*:$u \in [M]_{v,n}$
hence ∗:*DBM-val-bounded v u M n* **by** (*simp add*: *DBM-zone-repr-def*)
note ∗∗ = *DBM-val-bounded-neg-cycle*[*OF this - - - surj-on*]
have *cyc-free*: *cyc-free M n* **using** ∗∗ **by** *fastforce*
from *cyc-free-diag*[*OF this*] ‹$k \leq n$› **have** *M k k* $\geq$ *0* **by** *auto*

have *DBM-val-bounded v u* (*FWI M n k*) *n* **unfolding** *DBM-val-bounded-def*
proof (*safe*, *goal-cases*)

case *1*
with ‹$k \le n$› ‹$M\ k\ k \ge 0$› *cyc-free* **show** *?case*
unfolding *FWI-def neutral*[*symmetric*] *less-eq*[*symmetric*]
by − (*rule fwi-cyc-free-diag*[**where** $I = \{0..n\}$]; *auto*)
next
case (*2 c*)
with ‹$k \le n$› ‹$M\ k\ k \ge 0$› *fwi-len-distinct*[*of 0 n v c k M*] **obtain** *xs* **where** *xs*:
FWI M n k 0 (v c) = len M 0 (v c) xs set xs $\subseteq \{0..n\}$ *0* $\notin$ *set xs*
unfolding *FWI-def* **by** *force*
with *surj-on* ‹$v\ c \le n$› **show** *?case* **unfolding** *xs*(*1*)
by − (*rule DBM-val-bounded-len'2*[*OF* ∗]; *auto*)
next
case (*3 c*)
with ‹$k \le n$› ‹$M\ k\ k \ge 0$› *fwi-len-distinct*[*of v c n 0 k M*] **obtain** *xs* **where** *xs*:
FWI M n k (v c) 0 = len M (v c) 0 xs set xs $\subseteq \{0..n\}$
0 $\notin$ *set xs v c* $\notin$ *set xs*
unfolding *FWI-def* **by** *force*
with *surj-on* ‹$v\ c \le n$› **show** *?case* **unfolding** *xs*(*1*)
by − (*rule DBM-val-bounded-len'1*[*OF* ∗]; *auto*)
next
case (*4 c1 c2*)
with ‹$k \le n$› ‹$M\ k\ k \ge 0$› *fwi-len-distinct*[*of v c1 n v c2 k M*] **obtain** *xs* **where** *xs*:
FWI M n k (v c1) (v c2) = len M (v c1) (v c2) xs set xs $\subseteq \{0..n\}$
v c1 $\notin$ *set xs v c2* $\notin$ *set xs distinct xs*
unfolding *FWI-def* **by** *force*
with *surj-on* ‹$v\ c1 \le n$› ‹$v\ c2 \le n$› **show** *?case*
unfolding *xs*(*1*) **by** − (*rule DBM-val-bounded-len'3*[*OF* ∗]; *auto dest*: *distinct-cnt*[*of - 0*])
qed
then show $u \in [FWI\ M\ n\ k]_{v,n}$ **unfolding** *DBM-zone-repr-def* **by** *simp*
qed

end

3 DBM Operations

theory *DBM-Operations*
imports
DBM-Basics
begin

3.1 Auxiliary

lemmas [*trans*] = *finite-subset*

lemma *finite-vimageI2*: *finite* (*h* −' *F*) **if** *finite F inj-on h* {*x*. *h x* ∈ *F*}
proof −
  have *h* −' *F* = *h* −' *F* ∩ {*x*. *h x* ∈ *F*}
    by *auto*
  from *that* **show** *?thesis*
    by(*subst* ‹*h* −' *F* = -›) (*rule finite-vimage-IntI*[*of F h* {*x*. *h x* ∈ *F*}])
qed

lemma *gt-swap*:
  fixes *a b c* :: *'t* :: *time*
  assumes *c* < *a* + *b*
  shows *c* < *b* + *a*
by (*simp add*: *add.commute assms*)

lemma *le-swap*:
  fixes *a b c* :: *'t* :: *time*
  assumes *c* ≤ *a* + *b*
  shows *c* ≤ *b* + *a*
by (*simp add*: *add.commute assms*)

abbreviation *clock-numbering* :: (*'c* ⇒ *nat*) ⇒ *bool*
where
  clock-numbering v ≡ ∀ *c*. *v c* > *0*

lemma *DBM-triv*:
  $u \vdash_{v,n} (\lambda i\ j.\ \infty)$
  unfolding *DBM-val-bounded-def* **by** (*auto simp*: *dbm-le-def*)

3.2 Relaxation

Relaxation of upper bound constraints on all variables. Used to compute time lapse in timed automata.

definition
  up :: (*'t*::*linordered-cancel-ab-semigroup-add*) *DBM* ⇒ *'t DBM*
where
  up M ≡
    λ *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*
using *assms*
proof (*cases*, *goal-cases*)
case *1* **thus** *?case* **by** (*cases*, *auto*)
next
case *2* **thus** *?case* **by** (*cases*, *auto*)
qed *auto*

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*
using *dbm-entry-val-mono2*[*folded less-eq*, *OF assms*] **by** *auto*

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*
using *dbm-entry-val-mono3*[*folded less-eq*, *OF assms*] **by** *auto*

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*
using *dbm-entry-val-mono1*[*folded less-eq*, *OF assms*] **by** *auto*

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*
using *dbm-entry-val-mono3*[*folded less-eq*, *OF assms*] **by** *auto*

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*
using *dbm-entry-val-mono2*[*folded less-eq*, *OF assms*] **by** *auto*

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*
using *dbm-entry-val-mono1*[*folded less-eq*, *OF assms*] **by** *auto*

lemma *DBM-up-complete'*: *clock-numbering v* $\Longrightarrow u \in ([M]_{v,n})^{\uparrow} \Longrightarrow u \in [up\ M]_{v,n}$
unfolding *up-def DBM-zone-repr-def DBM-val-bounded-def zone-delay-def*
proof (*safe*, *goal-cases*)
case *prems*: (*2 u d c*)

hence ∗: *dbm-entry-val u None* (*Some c*) (*M 0* (*v c*)) **by** *auto*
thus *?case*
proof (*cases*, *goal-cases*)
case (*1 d′*)
have $-$ (*u c* + *d*) ≤ $-$ *u c* **using** ‹*d* ≥ *0*› **by** *simp*
with *1*(*2*) **have** $-$ (*u c* + *d*)≤ *d′* **by** (*blast intro*: *order.trans*)
thus *?case* **unfolding** *cval-add-def* **using** *1* **by** *fastforce*
next
case (*2 d′*)
have $-$ (*u c* + *d*) ≤ $-$ *u c* **using** ‹*d* ≥ *0*› **by** *simp*
with *2*(*2*) **have** $-$ (*u c* + *d*) < *d′* **by** (*blast intro*: *order-le-less-trans*)
thus *?case* **unfolding** *cval-add-def* **using** *2* **by** *fastforce*
qed *auto*
next
case *prems*: (*4 u d c1 c2*)
then have
dbm-entry-val u (*Some c1*) *None* (*M* (*v c1*) *0*) *dbm-entry-val u None* (*Some c2*) (*M 0* (*v c2*))
by *auto*
from *dbm-entry-val-add-4*[*OF this*] *prems* **have**
dbm-entry-val u (*Some c1*) (*Some c2*) (*min* (*dbm-add* (*M* (*v c1*) *0*) (*M 0* (*v c2*))) (*M* (*v c1*) (*v c2*)))
by (*auto split*: *split-min*)
with *prems*(*1*) **show** *?case*
by (*cases min* (*dbm-add* (*M* (*v c1*) *0*) (*M 0* (*v c2*))) (*M* (*v c1*) (*v c2*)), *auto simp*: *cval-add-def*)
qed *auto*

fun *theLe* :: (*′t*::*time*) *DBMEntry* ⇒ *′t* **where**
theLe (*Le d*) = *d* |
theLe (*Lt d*) = *d* |
theLe ∞ = *0*

lemma *DBM-up-sound′*:
assumes *clock-numbering′ v n u* ∈ $[up\ M]_{v,n}$
shows *u* ∈ $([M]_{v,n})^{\uparrow}$
proof −
obtain *S-Max-Le* **where** *S-Max-Le*:
S-Max-Le = {*d* − *u c* | *c d*. *0* < *v c* ∧ *v c* ≤ *n* ∧ *M* (*v c*) *0* = *Le d*}
by *auto*
obtain *S-Max-Lt* **where** *S-Max-Lt*:
S-Max-Lt = {*d* − *u c* | *c d*. *0* < *v c* ∧ *v c* ≤ *n* ∧ *M* (*v c*) *0* = *Lt d*}
by *auto*
obtain *S-Min-Le* **where** *S-Min-Le*:

S-Min-Le = {− *d* − *u c*| *c d*. *0* < *v c* ∧ *v c* ≤ *n* ∧ *M 0* (*v c*) = *Le d*}
by *auto*
obtain *S-Min-Lt* **where** *S-Min-Lt*:
S-Min-Lt = {− *d* − *u c* | *c d*. *0* < *v c* ∧ *v c* ≤ *n* ∧ *M 0* (*v c*) = *Lt d*}
by *auto*
have *finite* {*c*. *0* < *v c* ∧ *v c* ≤ *n*} (**is** *finite ?S*)
proof −
have *?S* ⊆ *v* −‘ {*1*..*n*}
by *auto*
also have *finite* . . .
using *assms*(*1*) **by** (*auto intro*!: *finite-vimageI2 inj-onI*)
finally show *?thesis* **.**
qed
then have ∀ *f*. *finite* {(*c*,*b*) | *c b*. *0* < *v c* ∧ *v c* ≤ *n* ∧ *f M* (*v c*) = *b*}
by *auto*
moreover have
∀ *f K*. {(*c*,*K d*) | *c d*. *0* < *v c* ∧ *v c* ≤ *n* ∧ *f M* (*v c*) = *K d*}
⊆ {(*c*,*b*) | *c b*. *0* < *v c* ∧ *v c* ≤ *n* ∧ *f M* (*v c*) = *b*}
by *auto*
ultimately have *1*:
∀ *f K*. *finite* {(*c*,*K d*) | *c d*. *0* < *v c* ∧ *v c* ≤ *n* ∧ *f M* (*v c*) = *K d*}
using *finite-subset*
by *fast*
have ∀ *f K*. *theLe o K* = *id* ⟶ *finite* {(*c*,*d*) | *c d*. *0* < *v c* ∧ *v c* ≤ *n* ∧ *f M* (*v c*) = *K d*}
proof (*safe*, *goal-cases*)
case *prems*: (*1 f K*)
then have (*c*, *d*) = (λ (*c*,*b*). (*c*, *theLe b*)) (*c*, *K d*) **for** *c* :: ′*a* **and** *d*
by (*simp add*: *pointfree-idE*)
then have
{(*c*,*d*) | *c d*. *0* < *v c* ∧ *v c* ≤ *n* ∧ *f M* (*v c*) = *K d*}
= (λ (*c*,*b*). (*c*, *theLe b*)) ‘ {(*c*,*K d*) | *c d*. *0* < *v c* ∧ *v c* ≤ *n* ∧ *f M* (*v c*) = *K d*}
by (*force simp*: *split-beta*)
moreover from *1* **have**
finite ((λ (*c*,*b*). (*c*, *theLe b*)) ‘ {(*c*,*K d*) | *c d*. *0* < *v c* ∧ *v c* ≤ *n* ∧ *f M* (*v c*) = *K d*})
by *auto*
ultimately show *?case* **by** *auto*
qed
then have *finI*:
⋀ *f g K*. *theLe o K* = *id* ⟹ *finite* (*g* ‘ {(*c*,*d*) | *c d*. *0* < *v c* ∧ *v c* ≤ *n* ∧ *f M* (*v c*) = *K d*})
by *auto*

have
finite $((\lambda(c,d).\ -\ d\ -\ u\ c)$ ‘ $\{(c,d) \mid c\ d.\ 0 < v\ c \wedge v\ c \leq n \wedge M\ 0\ (v\ c) = Le\ d\})$
by (*rule finI*, *auto*)
moreover have
S-Min-Le $= ((\lambda(c,d).\ -\ d\ -\ u\ c)$ ‘ $\{(c,d) \mid c\ d.\ 0 < v\ c \wedge v\ c \leq n \wedge M\ 0\ (v\ c) = Le\ d\})$
using *S-Min-Le* **by** *auto*
ultimately have *fin-min-le*: *finite S-Min-Le* **by** *auto*

have
finite $((\lambda(c,d).\ -\ d\ -\ u\ c)$ ‘ $\{(c,d) \mid c\ d.\ 0 < v\ c \wedge v\ c \leq n \wedge M\ 0\ (v\ c) = Lt\ d\})$
by (*rule finI*, *auto*)
moreover have
S-Min-Lt $= ((\lambda(c,d).\ -\ d\ -\ u\ c)$ ‘ $\{(c,d) \mid c\ d.\ 0 < v\ c \wedge v\ c \leq n \wedge M\ 0\ (v\ c) = Lt\ d\})$
using *S-Min-Lt* **by** *auto*
ultimately have *fin-min-lt*: *finite S-Min-Lt* **by** *auto*

have *finite* $((\lambda(c,d).\ d\ -\ u\ c)$ ‘ $\{(c,d) \mid c\ d.\ 0 < v\ c \wedge v\ c \leq n \wedge M\ (v\ c)\ 0 = Le\ d\})$
by (*rule finI*, *auto*)
moreover have
S-Max-Le $= ((\lambda(c,d).\ d\ -\ u\ c)$ ‘ $\{(c,d) \mid c\ d.\ 0 < v\ c \wedge v\ c \leq n \wedge M\ (v\ c)\ 0 = Le\ d\})$
using *S-Max-Le* **by** *auto*
ultimately have *fin-max-le*: *finite S-Max-Le* **by** *auto*

have
finite $((\lambda(c,d).\ d\ -\ u\ c)$ ‘ $\{(c,d) \mid c\ d.\ 0 < v\ c \wedge v\ c \leq n \wedge M\ (v\ c)\ 0 = Lt\ d\})$
by (*rule finI*, *auto*)
moreover have
S-Max-Lt $= ((\lambda(c,d).\ d\ -\ u\ c)$ ‘ $\{(c,d) \mid c\ d.\ 0 < v\ c \wedge v\ c \leq n \wedge M\ (v\ c)\ 0 = Lt\ d\})$
using *S-Max-Lt* **by** *auto*
ultimately have *fin-max-lt*: *finite S-Max-Lt* **by** *auto*

{ fix x **assume** $x \in$ *S-Min-Le*
hence $x \leq 0$ **unfolding** *S-Min-Le*
proof (*safe*, *goal-cases*)
case (*1 c d*)

with *assms* **have** $-$ *u c* $\leq$ *d* **unfolding** *DBM-zone-repr-def DBM-val-bounded-def up-def* **by** *auto*
thus *?case* **by** (*simp add*: *minus-le-iff*)
qed
} **note** *Min-Le-le-0* = *this*
have *Min-Lt-le-0*: *x* < *0* **if** *x* ∈ *S-Min-Lt* **for** *x* **using** *that* **unfolding** *S-Min-Lt*
proof (*safe*, *goal-cases*)
case (*1 c d*)
with *assms* **have** $-$ *u c* < *d* **unfolding** *DBM-zone-repr-def DBM-val-bounded-def up-def* **by** *auto*
thus *?case* **by** (*simp add*: *minus-less-iff*)
qed

The following basically all use the same proof. Only the first is not completely identical but nearly identical.

{ **fix** *l r* **assume** *l* ∈ *S-Min-Le r* ∈ *S-Max-Le*
with *S-Min-Le S-Max-Le* **have** *l* ≤ *r*
proof (*safe*, *goal-cases*)
case (*1 c c′ d d′*)
note *G1* = *this*
hence ∗:(*up M*) (*v c′*) (*v c*) = *min* (*dbm-add* (*M* (*v c′*) *0*) (*M 0* (*v c*))) (*M* (*v c′*) (*v c*))
using *assms* **unfolding** *up-def* **by** (*auto split*: *split-min*)
have *dbm-entry-val u* (*Some c′*) (*Some c*) ((*up M*) (*v c′*) (*v c*))
using *assms G1* **unfolding** *DBM-zone-repr-def DBM-val-bounded-def* **by** *fastforce*
hence *dbm-entry-val u* (*Some c′*) (*Some c*) (*dbm-add* (*M* (*v c′*) *0*) (*M 0* (*v c*)))
using *dbm-entry-dbm-min′* ∗ **by** *auto*
hence *u c′* − *u c* ≤ *d′* + *d* **using** *G1* **by** *auto*
hence *u c′* + (− *u c* − *d*) ≤ *d′* **by** (*simp add*: *add-diff-eq diff-le-eq*)
hence − *u c* − *d* ≤ *d′* − *u c′* **by** (*simp add*: *add.commute le-diff-eq*)
thus *?case* **by** (*metis add-uminus-conv-diff uminus-add-conv-diff*)
qed
} **note** *EE* = *this*
{ **fix** *l r* **assume** *l* ∈ *S-Min-Le r* ∈ *S-Max-Le*
with *S-Min-Le S-Max-Le* **have** *l* ≤ *r*
proof (*safe*, *goal-cases*)
case (*1 c c′ d d′*)
note *G1* = *this*
hence ∗:(*up M*) (*v c′*) (*v c*) = *min* (*dbm-add* (*M* (*v c′*) *0*) (*M 0* (*v c*))) (*M* (*v c′*) (*v c*))
using *assms* **unfolding** *up-def* **by** (*auto split*: *split-min*)

have *dbm-entry-val u* (*Some c′*) (*Some c*) ((*up M*) (*v c′*) (*v c*))
using *assms G1* **unfolding** *DBM-zone-repr-def DBM-val-bounded-def* **by** *fastforce*
hence *dbm-entry-val u* (*Some c′*) (*Some c*) (*dbm-add* (*M* (*v c′*) *0*) (*M* *0* (*v c*)))
using *dbm-entry-dbm-min′* ∗ **by** *auto*
hence $u\ c' - u\ c \leq d' + d$ **using** *G1* **by** *auto*
hence $u\ c' + (-\ u\ c - d) \leq d'$ **by** (*simp add*: *add-diff-eq diff-le-eq*)
hence $-\ u\ c - d \leq d' - u\ c'$ **by** (*simp add*: *add.commute le-diff-eq*)
thus *?case* **by** (*metis add-uminus-conv-diff uminus-add-conv-diff*)
qed
} **note** *EE* = *this*
{ **fix** *l r* **assume** *l* ∈ *S-Min-Lt r* ∈ *S-Max-Le*
with *S-Min-Lt S-Max-Le* **have** *l* < *r*
proof (*safe*, *goal-cases*)
case (*1 c c′ d d′*)
note *G1* = *this*
hence ∗:(*up M*) (*v c′*) (*v c*) = *min* (*dbm-add* (*M* (*v c′*) *0*) (*M* *0* (*v* *c*))) (*M* (*v c′*) (*v c*))
using *assms* **unfolding** *up-def* **by** (*auto split*: *split-min*)
have *dbm-entry-val u* (*Some c′*) (*Some c*) ((*up M*) (*v c′*) (*v c*))
using *assms G1* **unfolding** *DBM-zone-repr-def DBM-val-bounded-def* **by** *fastforce*
hence *dbm-entry-val u* (*Some c′*) (*Some c*) (*dbm-add* (*M* (*v c′*) *0*) (*M* *0* (*v c*)))
using *dbm-entry-dbm-min′* ∗ **by** *auto*
hence $u\ c' - u\ c < d' + d$ **using** *G1* **by** *auto*
hence $u\ c' + (-\ u\ c - d) < d'$ **by** (*simp add*: *add-diff-eq diff-less-eq*)
hence $-\ u\ c - d < d' - u\ c'$ **by** (*simp add*: *add.commute less-diff-eq*)
thus *?case* **by** (*metis add-uminus-conv-diff uminus-add-conv-diff*)
qed
} **note** *LE* = *this*
{ **fix** *l r* **assume** *l* ∈ *S-Min-Le r* ∈ *S-Max-Lt*
with *S-Min-Le S-Max-Lt* **have** *l* < *r*
proof (*safe*, *goal-cases*)
case (*1 c c′ d d′*)
note *G1* = *this*
hence ∗:(*up M*) (*v c′*) (*v c*) = *min* (*dbm-add* (*M* (*v c′*) *0*) (*M* *0* (*v* *c*))) (*M* (*v c′*) (*v c*))
using *assms* **unfolding** *up-def* **by** (*auto split*: *split-min*)
have *dbm-entry-val u* (*Some c′*) (*Some c*) ((*up M*) (*v c′*) (*v c*))
using *assms G1* **unfolding** *DBM-zone-repr-def DBM-val-bounded-def* **by** *fastforce*
hence *dbm-entry-val u* (*Some c′*) (*Some c*) (*dbm-add* (*M* (*v c′*) *0*) (*M*

0 (*v c*)))
  using *dbm-entry-dbm-min′* ∗ **by** *auto*
  hence $u\ c' - u\ c < d' + d$ **using** *G1* **by** *auto*
  hence $u\ c' + (- u\ c - d) < d'$ **by** (*simp add*: *add-diff-eq diff-less-eq*)
  hence $- u\ c - d < d' - u\ c'$ **by** (*simp add*: *add.commute less-diff-eq*)
  thus *?case* **by** (*metis add-uminus-conv-diff uminus-add-conv-diff*)
 qed
} **note** *EL* = *this*
{ **fix** *l r* **assume** $l \in$ *S-Min-Lt* $r \in$ *S-Max-Lt*
 with *S-Min-Lt S-Max-Lt* **have** $l < r$
 proof (*safe*, *goal-cases*)
  case (*1 c c′ d d′*)
  note *G1* = *this*
  hence ∗:(*up M*) (*v c′*) (*v c*) = *min* (*dbm-add* (*M* (*v c′*) *0*) (*M 0* (*v c*))) (*M* (*v c′*) (*v c*))
  using *assms* **unfolding** *up-def* **by** (*auto split*: *split-min*)
  have *dbm-entry-val u* (*Some c′*) (*Some c*) ((*up M*) (*v c′*) (*v c*))
  using *assms G1* **unfolding** *DBM-zone-repr-def DBM-val-bounded-def* **by** *fastforce*
  hence *dbm-entry-val u* (*Some c′*) (*Some c*) (*dbm-add* (*M* (*v c′*) *0*) (*M 0* (*v c*)))
  using *dbm-entry-dbm-min′* ∗ **by** *auto*
  hence $u\ c' - u\ c < d' + d$ **using** *G1* **by** *auto*
  hence $u\ c' + (- u\ c - d) < d'$ **by** (*simp add*: *add-diff-eq diff-less-eq*)
  hence $- u\ c - d < d' - u\ c'$ **by** (*simp add*: *add.commute less-diff-eq*)
  thus *?case* **by** (*metis add-uminus-conv-diff uminus-add-conv-diff*)
 qed
} **note** *LL* = *this*
obtain *m* **where** *m*: $\forall\ t \in$ *S-Min-Le*. $m \geq t$ $\forall\ t \in$ *S-Min-Lt*. $m > t$
 $\forall\ t \in$ *S-Max-Le*. $m \leq t$ $\forall\ t \in$ *S-Max-Lt*. $m < t$ $m \leq 0$
proof −
 assume *m*:($\bigwedge m.\ \forall t\in$*S-Min-Le*. $t \leq m \Longrightarrow$
  $\forall t\in$*S-Min-Lt*. $t < m \Longrightarrow \forall t\in$*S-Max-Le*. $m \leq t \Longrightarrow \forall t\in$*S-Max-Lt*. $m < t \Longrightarrow m \leq 0 \Longrightarrow$ *thesis*)
 let *?min-le* = *Max S-Min-Le*
 let *?min-lt* = *Max S-Min-Lt*
 let *?max-le* = *Min S-Max-Le*
 let *?max-lt* = *Min S-Max-Lt*
 show *thesis*
 proof (*cases S-Min-Le* = {} ∧ *S-Min-Lt* = {})
  case *True*
  note *T* = *this*
  show *thesis*
  proof (*cases S-Max-Le* = {} ∧ *S-Max-Lt* = {})

```
        case True
        let ?d' = 0 :: 't :: time
        show thesis using True T by (intro m[of ?d']) auto
      next
        case False
        let ?d =
          if S-Max-Le ≠ {}
           then if S-Max-Lt ≠ {} then min ?max-lt ?max-le else ?max-le
           else ?max-lt
        obtain a :: 'b where a: a < 0 using non-trivial-neg by auto
        let ?d' = min 0 (?d + a)
        { fix x assume x ∈ S-Max-Le
          with fin-max-le a have min 0 (Min S-Max-Le + a) ≤ x
               by (metis Min-le add-le-same-cancel1 le-less-trans less-imp-le
min.coboundedI2 not-less)
          then have min 0 (Min S-Max-Le + a) ≤ x by auto
        } note 1 = this
        { fix x assume x: x ∈ S-Max-Lt
         have min 0 (min (Min S-Max-Lt) (Min S-Max-Le) + a) < ?max-lt
        by (meson a add-less-same-cancel1 min.coboundedI min.strict-coboundedI2
order.strict-trans2)
          also from fin-max-lt x have ... ≤ x by auto
         finally have min 0 (min (Min S-Max-Lt) (Min S-Max-Le) + a) <
x .
        } note 2 = this
        { fix x assume x: x ∈ S-Max-Le
         have min 0 (min (Min S-Max-Lt) (Min S-Max-Le) + a) ≤ ?max-le
            by (metis le-add-same-cancel1 linear not-le a min-le-iff-disj)
          also from fin-max-le x have ... ≤ x by auto
         finally have min 0 (min (Min S-Max-Lt) (Min S-Max-Le) + a) ≤
x .
        } note 3 = this
        show thesis using False T a 1 2 3
          apply (intro m[of ?d'])
          apply simp-all
       apply (metis Min.coboundedI add-less-same-cancel1 dual-order.strict-trans2
fin-max-lt
              min.boundedE not-le)
          done
      qed
    next
      case False
      note F = this
      show thesis
```

proof (*cases S-Max-Le* = {} ∧ *S-Max-Lt* = {})
case *True*
let *?d′* = *0* :: *′t* :: *time*
show *thesis* **using** *True Min-Le-le-0 Min-Lt-le-0* **by** (*intro m*[*of ?d′*]) *auto*
next
case *False*
let *?r* =
if S-Max-Le ≠ {}
then if S-Max-Lt ≠ {} *then min ?max-lt ?max-le else ?max-le*
else ?max-lt
let *?l* =
if S-Min-Le ≠ {}
then if S-Min-Lt ≠ {} *then max ?min-lt ?min-le else ?min-le*
else ?min-lt

have *1*: *x* ≤ *max ?min-lt ?min-le x* ≤ *?min-le* **if** *x* ∈ *S-Min-Le* **for** *x*
using *that fin-min-le* **by** (*simp add*: *max.coboundedI2*)+

{
fix *x y* **assume** *x*: *x* ∈ *S-Max-Le y* ∈ *S-Min-Lt*
then have *S-Min-Lt* ≠ {} **by** *auto*
from *LE*[*OF Max-in*[*OF fin-min-lt*], *OF this*, *OF x*(*1*)] **have** *?min-lt* ≤ *x* **by** *auto*
} **note** *3* = *this*

have *4*: *?min-le* ≤ *x* **if** *x* ∈ *S-Max-Le y* ∈ *S-Min-Le* **for** *x y*
using *EE*[*OF Max-in*[*OF fin-min-le*], *OF - that*(*1*)] *that* **by** *auto*

{
fix *x y* **assume** *x*: *x* ∈ *S-Max-Lt y* ∈ *S-Min-Lt*
then have *S-Min-Lt* ≠ {} **by** *auto*
from *LL*[*OF Max-in*[*OF fin-min-lt*], *OF this*, *OF x*(*1*)] **have** *?min-lt* < *x* **by** *auto*
} **note** *5* = *this*
{
fix *x y* **assume** *x*: *x* ∈ *S-Max-Lt y* ∈ *S-Min-Le*
then have *S-Min-Le* ≠ {} **by** *auto*
from *EL*[*OF Max-in*[*OF fin-min-le*], *OF this*, *OF x*(*1*)] **have** *?min-le* < *x* **by** *auto*
} **note** *6* = *this*
{
fix *x y* **assume** *x*: *y* ∈ *S-Min-Le*
then have *S-Min-Le* ≠ {} **by** *auto*

```
      from Min-Le-le-0[OF Max-in[OF fin-min-le], OF this] have ?min-le
≤ 0 by auto
      } note 7 = this
      {
       fix x y assume x: y ∈ S-Min-Lt
       then have S-Min-Lt ≠ {} by auto
       from Min-Lt-le-0[OF Max-in[OF fin-min-lt], OF this] have ?min-lt
< 0 ?min-lt ≤ 0 by auto
      } note 8 = this
      show thesis
      proof (cases ?l < ?r)
       case False
       then have *: S-Max-Le ≠ {}
       proof (safe, goal-cases)
        case 1
         with ‹¬ (S-Max-Le = {} ∧ S-Max-Lt = {})› obtain y where
y:y ∈ S-Max-Lt by auto
        note 1 = 1 this
        { fix x y assume A: x ∈ S-Min-Le y ∈ S-Max-Lt
          with EL[OF Max-in[OF fin-min-le] Min-in[OF fin-max-lt]]
          have Max S-Min-Le < Min S-Max-Lt by auto
        } note ** = this
        { fix x y assume A: x ∈ S-Min-Lt y ∈ S-Max-Lt
          with LL[OF Max-in[OF fin-min-lt] Min-in[OF fin-max-lt]]
          have Max S-Min-Lt < Min S-Max-Lt by auto
        } note *** = this
        show ?case
        proof (cases S-Min-Le ≠ {})
          case True
          note T = this
          show ?thesis
          proof (cases S-Min-Lt ≠ {})
            case True
            then show False using 1 T True ** *** by auto
          next
            case False with 1 T ** show False by auto
          qed
        next
          case False
          with 1 False *** ‹¬ (S-Min-Le = {} ∧ S-Min-Lt = {})› show
?thesis by auto
        qed
       qed
       { fix x y assume A: x ∈ S-Min-Lt y ∈ S-Max-Lt
```

```
      with LL[OF Max-in[OF fin-min-lt] Min-in[OF fin-max-lt]]
      have Max S-Min-Lt < Min S-Max-Lt by auto
    } note *** = this
    { fix x y assume A: x ∈ S-Min-Lt y ∈ S-Max-Le
      with LE[OF Max-in[OF fin-min-lt] Min-in[OF fin-max-le]]
      have Max S-Min-Lt < Min S-Max-Le by auto
    } note **** = this
    from F False have **: S-Min-Le ≠ {}
    proof (safe, goal-cases)
      case (1 x)
      show ?case
      proof (cases S-Max-Le ≠ {})
        case True
        note T = this
        show ?thesis
        proof (cases S-Max-Lt ≠ {})
          case True
          then show x ∈ {} using 1 T True **** *** by auto
        next
          case False with 1 T **** show x ∈ {} by auto
        qed
      next
        case False
        with 1 False *** ‹¬ (S-Max-Le = {} ∧ S-Max-Lt = {})› show
?thesis by auto
      qed
    qed
    {
      fix x assume x: x ∈ S-Min-Lt
          then have x ≤ ?min-lt using fin-min-lt by (simp add:
max.coboundedI2)
      also have ?min-lt < ?min-le
      proof (rule ccontr, goal-cases)
        case 1
        with x ** have 1: ?l = ?min-lt by auto
        have 2: ?min-lt < ?max-le using * ****[OF x] by auto
        show False
        proof (cases S-Max-Lt = {})
          case False
          then have ?min-lt < ?max-lt using * ***[OF x] by auto
          with 1 2 have ?l < ?r by auto
          with ‹¬ ?l < ?r› show False by auto
        next
          case True
```

with *1 2* **have** *?l* < *?r* **by** *auto*
with ‹¬ *?l* < *?r*› **show** *False* **by** *auto*
qed
qed
finally have *x* < *max ?min-lt ?min-le* **by** (*simp add*: *max.strict-coboundedI2*)
} **note** *2* = *this*
show *thesis* **using** *F False 1 2 3 4 5 6 7 8* ∗ ∗∗ **by** ((*intro m*[*of ?l*]), *auto*)
next
case *True*
then obtain *d* **where** *d*: *?l* < *d d* < *?r* **using** *dense* **by** *auto*
let *?d′* = *min 0 d*
{
fix *t* **assume** *t* ∈ *S-Min-Le*
then have *t* ≤ *?l* **using** *1* **by** *auto*
with *d* **have** *t* ≤ *d* **by** *auto*
}
moreover {
fix *t* **assume** *t*: *t* ∈ *S-Min-Lt*
then have *t* ≤ *max ?min-lt ?min-le* **using** *fin-min-lt* **by** (*simp add*: *max.coboundedI1*)
with *t Min-Lt-le-0* **have** *t* ≤ *?l* **using** *fin-min-lt* **by** *auto*
with *d* **have** *t* < *d* **by** *auto*
}
moreover {
fix *t* **assume** *t*: *t* ∈ *S-Max-Le*
then have *min ?max-lt ?max-le* ≤ *t* **using** *fin-max-le* **by** (*simp add*: *min.coboundedI2*)
then have *?r* ≤ *t* **using** *fin-max-le t* **by** *auto*
with *d* **have** *d* ≤ *t* **by** *auto*
then have *min 0 d* ≤ *t* **by** (*simp add*: *min.coboundedI2*)
}
moreover {
fix *t* **assume** *t*: *t* ∈ *S-Max-Lt*
then have *min ?max-lt ?max-le* ≤ *t* **using** *fin-max-lt* **by** (*simp add*: *min.coboundedI1*)
then have *?r* ≤ *t* **using** *fin-max-lt t* **by** *auto*
with *d* **have** *d* < *t* **by** *auto*
then have *min 0 d* < *t* **by** (*simp add*: *min.strict-coboundedI2*)
}
ultimately show *thesis* **using** *Min-Le-le-0 Min-Lt-le-0* **by** ((*intro m*[*of ?d′*]), *auto*)
qed
qed

qed
qed
obtain u' **where** $u' = (u \oplus m)$ **by** *blast*
hence u': $u = (u' \oplus (-m))$ **unfolding** *cval-add-def* **by** *force*
have *DBM-val-bounded v* u' *M n* **unfolding** *DBM-val-bounded-def*
proof (*safe*, *goal-cases*)
case *1* **with** *assms*(*1*,*2*) **show** *?case* **unfolding** *DBM-zone-repr-def DBM-val-bounded-def up-def* **by** *auto*
next
case (*3 c*)
thus *?case*
proof (*cases M* (*v c*) *0*, *goal-cases*)
case (*1 x1*)
hence $m \leq x1 - u\ c$ **using** *m*(*3*) *S-Max-Le assms* **by** *auto*
hence $u\ c + m \leq x1$ **by** (*simp add*: *add.commute le-diff-eq*)
thus *?case* **using** u' *1*(*2*) **unfolding** *cval-add-def* **by** *auto*
next
case (*2 x2*)
hence $m < x2 - u\ c$ **using** *m*(*4*) *S-Max-Lt assms* **by** *auto*
hence $u\ c + m < x2$ **by** (*metis add-less-cancel-left diff-add-cancel gt-swap*)
thus *?case* **using** u' *2*(*2*) **unfolding** *cval-add-def* **by** *auto*
next
case *3* **thus** *?case* **by** *auto*
qed
next
case (*2 c*) **thus** *?case*
proof (*cases M 0* (*v c*), *goal-cases*)
case (*1 x1*)
hence $- x1 - u\ c \leq m$ **using** *m*(*1*) *S-Min-Le assms* **by** *auto*
hence $- u\ c - m \leq x1$ **using** *diff-le-eq neg-le-iff-le* **by** *fastforce*
thus *?case* **using** u' *1*(*2*) **unfolding** *cval-add-def* **by** *auto*
next
case (*2 x2*)
hence $- x2 - u\ c < m$ **using** *m*(*2*) *S-Min-Lt assms* **by** *auto*
hence $- u\ c - m < x2$ **using** *diff-less-eq neg-less-iff-less* **by** *fastforce*
thus *?case* **using** u' *2*(*2*) **unfolding** *cval-add-def* **by** *auto*
next
case *3* **thus** *?case* **by** *auto*
qed
next
case (*4 c1 c2*)
from *assms* **have** $v\ c1 > 0$ $v\ c2 \neq 0$ **by** *auto*
then have *B*: (*up M*) (*v c1*) (*v c2*) = *min* (*dbm-add* (*M* (*v c1*) *0*) (*M*

0 (*v c2*))) (*M* (*v c1*) (*v c2*))
unfolding *up-def* **by** *simp*

show *?case*
proof (*cases* (*dbm-add* (*M* (*v c1*) *0*) (*M* *0* (*v c2*))) < (*M* (*v c1*) (*v c2*)))
case *False*
with *B* **have** (*up M*) (*v c1*) (*v c2*) = *M* (*v c1*) (*v c2*) **by** (*auto split*: *split-min*)
with *assms 4* **have**
dbm-entry-val u (*Some c1*) (*Some c2*) (*M* (*v c1*) (*v c2*))
unfolding *DBM-zone-repr-def* **unfolding** *DBM-val-bounded-def* **by** *fastforce*
thus *?thesis* **using** *u′* **by** *cases* (*auto simp add*: *cval-add-def*)
next
case *True*
with *B* **have** (*up M*) (*v c1*) (*v c2*) = *dbm-add* (*M* (*v c1*) *0*) (*M* *0* (*v c2*)) **by** (*auto split*: *split-min*)
with *assms 4* **have**
dbm-entry-val u (*Some c1*) (*Some c2*) (*dbm-add* (*M* (*v c1*) *0*) (*M* *0* (*v c2*)))
unfolding *DBM-zone-repr-def* **unfolding** *DBM-val-bounded-def* **by** *fastforce*
with *True dbm-entry-dbm-lt* **have**
dbm-entry-val u (*Some c1*) (*Some c2*) (*M* (*v c1*) (*v c2*))
unfolding *less* **by** *fast*
thus *?thesis* **using** *u′* **by** *cases* (*auto simp add*: *cval-add-def*)
qed
qed
with *m*(*5*) *u′* **show** *?thesis*
unfolding *DBM-zone-repr-def zone-delay-def* **by** *fastforce*
qed

3.3 Intersection

fun *And* :: (*′t* :: {*linordered-cancel-ab-monoid-add*}) *DBM* ⇒ *′t DBM* ⇒ *′t DBM* **where**
And M1 M2 = (λ *i j*. *min* (*M1 i j*) (*M2 i j*))

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*
using *assms* **unfolding** *DBM-val-bounded-def* **by** (*auto simp*: *min-def*)

lemma *DBM-and-sound1*:
  assumes *DBM-val-bounded v u* (*And M1 M2*) *n*
  shows *DBM-val-bounded v u M1 n*
  using *assms* **unfolding** *DBM-val-bounded-def*
  apply *safe*
    apply (*simp add*: *less-eq*[*symmetric*]; *fail*)
   apply (*auto 4 3 intro*: *dbm-entry-val-mono*[*folded less-eq*])
  done

lemma *DBM-and-sound2*:
  assumes *DBM-val-bounded v u* (*And M1 M2*) *n*
  shows *DBM-val-bounded v u M2 n*
  using *assms* **unfolding** *DBM-val-bounded-def*
  apply *safe*
    apply (*simp add*: *less-eq*[*symmetric*]; *fail*)
   apply (*auto 4 3 intro*: *dbm-entry-val-mono*[*folded less-eq*])
  done

lemma *And-correct*:
  $[M1]_{v,n} \cap [M2]_{v,n} = [And\ M1\ M2]_{v,n}$
 using *DBM-and-sound1 DBM-and-sound2 DBM-and-complete* **unfolding** *DBM-zone-repr-def* **by** *blast*

3.4 Variable Reset

definition
  $DBM\text{-}reset :: ('t :: time)\ DBM \Rightarrow nat \Rightarrow nat \Rightarrow 't \Rightarrow 't\ DBM \Rightarrow bool$
where
  $DBM\text{-}reset\ M\ n\ k\ d\ M' \equiv$
    $(\forall\ j \leq n.\ 0 < j \wedge k \neq j \longrightarrow M'\ k\ j = \infty \wedge M'\ j\ k = \infty) \wedge M'\ k\ 0 = Le\ d \wedge M'\ 0\ k = Le\ (-\ d)$
    $\wedge\ M'\ k\ k = M\ k\ k$
    $\wedge\ (\forall i \leq n.\ \forall j \leq n.$
      $i \neq k \wedge j \neq k \longrightarrow M'\ i\ j = min\ (dbm\text{-}add\ (M\ i\ k)\ (M\ k\ j))\ (M\ i\ j))$

lemma *DBM-reset-mono*:
  assumes $DBM\text{-}reset\ M\ n\ k\ d\ M'\ i \leq n\ j \leq n\ i \neq k\ j \neq k$
  shows $M'\ i\ j \leq M\ i\ j$
using *assms* **unfolding** *DBM-reset-def* **by** *auto*

lemma *DBM-reset-len-mono*:
  assumes $DBM\text{-}reset\ M\ n\ k\ d\ M'\ k \notin set\ xs\ i \neq k\ j \neq k\ set\ (i\ \#\ j\ \#\ xs) \subseteq \{0..n\}$

shows *len M' i j xs* ≤ *len M i j xs*
using *assms* **by** (*induction xs arbitrary*: *i*) (*auto intro*: *add-mono DBM-reset-mono*)

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 (*cases xs* = [])
case *Nil*: *True*
show *?thesis*
proof (*cases k* = *i*)
case *True*
with *Nil assms* **have** *len M' i i* [] < *Le 0* **unfolding** *DBM-reset-def* **by** *auto*
moreover from *assms* **have** *set* (*i* # []) ⊆ {*0..n*} **by** *auto*
ultimately show *?thesis* **by** *blast*
next
case *False*
with *Nil assms DBM-reset-mono* **have** *len M' i i* [] < *Le 0* **by** *fastforce*
moreover from *assms* **have** *set* (*i* # []) ⊆ {*0..n*} **by** *auto*
ultimately show *?thesis* **by** *blast*
qed
next
case *False*
with *assms* **obtain** *j ys* **where** *cycle*:
len M j j ys < *Le 0 distinct* (*j* # *ys*) *j* ∈ *set* (*i* # *xs*) *set ys* ⊆ *set xs*
by (*metis negative-len-shortest neutral*)
show *?thesis*
proof (*cases k* ∈ *set* (*j* # *ys*))
case *False*
with *cycle assms* **have** *len M' j j ys* ≤ *len M j j ys* **by** − (*rule DBM-reset-len-mono*, *auto*)
moreover from *cycle assms* **have** *set* (*j* # *ys*) ⊆ {*0..n*} **by** *auto*
ultimately show *?thesis* **using** *cycle*(*1*) **by** *fastforce*
next
case *True*
then obtain *l* **where** *l*: (*l*, *k*) ∈ *set* (*arcs j j ys*)
proof (*cases j* = *k*, *goal-cases*)
case *True*
show *?thesis*
proof (*cases ys* = [])
case *T*: *True*
with *True* **show** *?thesis* **by** (*auto intro*: *that*)
next

case *False*
then obtain *z zs* **where** $ys = zs @ [z]$ **by** (*metis append-butlast-last-id*)
from *arcs-decomp*[*OF this*] *True* **show** *?thesis* **by** (*auto intro*: *that*)
qed
next
case *False*
from *arcs-set-elem2*[*OF False True*] **show** *?thesis* **by** (*blast intro*: *that*)
qed
show *?thesis*
proof (*cases* $ys = []$)
case *False*
from *cycle-rotate-2'*[*OF False l*, *of M*] *cycle*(*1*) **obtain** *zs* **where** *rotated*:
len M l l ($k \# zs$) $<$ *Le 0 set* ($l \# k \# zs$) $=$ *set* ($j \# ys$) $1 +$ *length zs* $=$ *length ys*
by *auto*
with *length-eq-distinct*[*OF this*(*2*)[*symmetric*] *cycle*(*2*)] **have** *distinct* ($l \# k \# zs$) **by** *auto*
note *rotated* $=$ *rotated*(*1*,*2*) *this*
from *this*(*2*) *cycle*(*3*,*4*) *assms*(*3*) **have** *n-bound*: *set* ($l \# k \# zs$) $\subseteq \{0..n\}$ **by** *auto*
then have $l \leq n$ **by** *auto*
show *?thesis*
proof (*cases zs*)
case *Nil*
with *rotated* **have** $M\ l\ k + M\ k\ l <$ *Le 0* $l \neq k$ **by** *auto*
with *assms*(*1*) ‹$l \leq n$› **have** $M'\ l\ l <$ *Le 0* **unfolding** *DBM-reset-def add min-def* **by** *auto*
with ‹$l \leq n$› **have** *len* $M'\ l\ l\ [] <$ *Le 0 set* $[l] \subseteq \{0..n\}$ **by** *auto*
then show *?thesis* **by** *blast*
next
case (*Cons w ws*)
with *n-bound* **have** $*$: *set* ($w \# l \# ws$) $\subseteq \{0..n\}$ **by** *auto*
from *Cons n-bound rotated*(*3*) **have** $w \leq n$ $w \neq k$ $l \neq k$ **by** *auto*
with *assms*(*1*) ‹$l \leq n$› **have**
$M'\ l\ w \leq M\ l\ k + M\ k\ w$
unfolding *DBM-reset-def add min-def* **by** *auto*
moreover from *Cons rotated assms* $*$ **have**
len M' *w l ws* $\leq$ *len M w l ws*
by $-$ (*rule DBM-reset-len-mono*, *auto*)
ultimately have
len M' *l l zs* $\leq$ *len M l l* ($k \# zs$)
using *Cons* **by** (*auto intro*: *add-mono simp add*: *add.assoc*[*symmetric*])
with *n-bound rotated*(*1*) **show** *?thesis* **by** *fastforce*

qed
next
case T: *True*
with *True cycle* **have** $M\ j\ j < Le\ 0\ j = k$ **by** *auto*
with *assms*(*1*) **have** *len* $M'\ k\ k\ [] < Le\ 0$ **unfolding** *DBM-reset-def* **by** *simp*
moreover from *assms*(*3*) **have** *set* $(k \# []) \subseteq \{0..n\}$ **by** *auto*
ultimately show *?thesis* **by** *blast*
qed
qed
qed

Implementation of DBM reset

definition
reset :: (*'t*::{*linordered-cancel-ab-semigroup-add*,*uminus*}) *DBM* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *'t* $\Rightarrow$ *'t DBM*
where
reset M n k d =
(λ *i j*.
if $i = k \wedge j = 0$ *then Le d*
else if $i = 0 \wedge j = k$ *then Le* $(-d)$
else if $i = k \wedge j \neq k$ *then* ∞
else if $i \neq k \wedge j = k$ *then* ∞
else if $i = k \wedge j = k$ *then M k k*
else min (*dbm-add* (*M i k*) (*M k j*)) (*M i j*)
)

fun
reset' ::
(*'t*::{*linordered-cancel-ab-semigroup-add*,*uminus*}) *DBM*
$\Rightarrow$ *nat* $\Rightarrow$ *'c list* $\Rightarrow$ (*'c* $\Rightarrow$ *nat*) $\Rightarrow$ *'t* $\Rightarrow$ *'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*:
$0 < k \Longrightarrow k \leq n \Longrightarrow$ *DBM-reset* $M\ n\ k\ d$ (*reset* $M\ n\ k\ d$)
unfolding *DBM-reset-def* **by** (*auto simp*: *reset-def*)

lemma *DBM-reset-complete*:
assumes *clock-numbering'* $v\ n\ v\ c \leq n$ *DBM-reset* $M\ n\ (v\ c)\ d\ M'$ *DBM-val-bounded* $v\ u\ M\ n$
shows *DBM-val-bounded* $v\ (u(c := d))\ M'\ n$
unfolding *DBM-val-bounded-def* **using** *assms*

proof (*safe*, *goal-cases*)
  case *1*
  then have ∗: *M 0 0* ≥ *Le 0* **unfolding** *DBM-val-bounded-def less-eq* **by** *auto*
  from *1* **have** ∗∗: *M′ 0 0* = *min* (*M 0* (*v c*) + *M* (*v c*) *0*) (*M 0 0*)
    unfolding *DBM-reset-def add* **by** *auto*
  show *?case*
  proof (*cases M 0* (*v c*) + *M* (*v c*) *0* ≤ *M 0 0*)
    case *False*
    with ∗ ∗∗ **show** *?thesis* **unfolding** *min-def less-eq* **by** *auto*
  next
    case *True*
    have *dbm-entry-val u* (*Some c*) (*Some c*) (*M* (*v c*) *0* + *M 0* (*v c*))
      by (*metis DBM-val-bounded-def assms*(*2*,*4*) *dbm-entry-val-add-4 add*)
    then have *M* (*v c*) *0* + *M 0* (*v c*) ≥ *Le 0*
      unfolding *less-eq dbm-le-def* **by** (*cases M* (*v c*) *0* + *M 0* (*v c*)) *auto*
    with *True* ∗∗ **have** *M′ 0 0* ≥ *Le 0* **by** (*simp add*: *comm*)
    then show *?thesis* **unfolding** *less-eq* **.**
  qed
next
  case (*2 c′*)
  show *?case*
  proof (*cases c* = *c′*)
    case *False*
    hence *F*:*v c′* ≠ *v c* **using** *2* **by** *metis*
    hence ∗:*M′ 0* (*v c′*) = *min* (*dbm-add* (*M 0* (*v c*)) (*M* (*v c*) (*v c′*))) (*M 0* (*v c′*))
    using *F 2* **unfolding** *DBM-reset-def* **by** *simp*
    show *?thesis*
    proof (*cases dbm-add* (*M 0* (*v c*)) (*M* (*v c*) (*v c′*)) < *M 0* (*v c′*))
      case *False*
      with ∗ **have** *M′ 0* (*v c′*) = *M 0* (*v c′*) **by** (*auto split*: *split-min*)
      hence *dbm-entry-val u None* (*Some c′*) (*M′ 0* (*v c′*))
      using *2* **unfolding** *DBM-val-bounded-def* **by** *auto*
      thus *?thesis* **using** *F* **by** *cases fastforce*+
    next
      case *True*
      with ∗ **have** ∗∗:*M′ 0* (*v c′*) = *dbm-add* (*M 0* (*v c*)) (*M* (*v c*) (*v c′*)) **by** (*auto split*: *split-min*)
      from *2* **have** ∗∗∗:*dbm-entry-val u None* (*Some c*) (*M 0* (*v c*))
        dbm-entry-val u (*Some c*) (*Some c′*) (*M* (*v c*) (*v c′*))
        unfolding *DBM-val-bounded-def* **by** *auto*
      show *?thesis*
      proof −

note ∗∗∗
moreover have *dbm-entry-val* $(u(c := d))$ *None* (*Some* c') (*dbm-add* (*Le d1*) (M (v c) (v c')))
if $M\ 0\ (v\ c) =$ *Le d1*
and *dbm-entry-val u* (*Some c*) (*Some* c') (M (v c) (v c'))
and $-\ u\ c \leq d1$
for *d1* :: $'b$
proof −
note $G1 = that$
from $G1(2)$ **show** *?thesis*
proof (*cases*, *goal-cases*)
case ($1\ d'$)
from ‹$u\ c - u\ c' \leq d'$› $G1(3)$ **have** $-\ u\ c' \leq d1 + d'$
by (*metis diff-minus-eq-add less-diff-eq less-le-trans minus-diff-eq minus-le-iff not-le*)
thus *?case* **using** *1* ‹$c \neq c'$› **by** *fastforce*
next
case ($2\ d'$)
from *this*(2) $G1(3)$ **have** $u\ c - u\ c' - u\ c < d1 + d'$ **using** *add-le-less-mono* **by** *fastforce*
hence $-\ u\ c' < d1 + d'$ **by** *simp*
thus *?case* **using** *2* ‹$c \neq c'$› **by** *fastforce*
next
case (*3*) **thus** *?case* **by** *auto*
qed
qed
moreover have *dbm-entry-val* $(u(c := d))$ *None* (*Some* c') (*dbm-add* (*Lt d2*) (M (v c) (v c')))
if $M\ 0\ (v\ c) =$ *Lt d2*
and *dbm-entry-val u* (*Some c*) (*Some* c') (M (v c) (v c'))
and $-\ u\ c < d2$
for *d2* :: $'b$
proof −
note $G2 = that$
from *this*(2) **show** *?thesis*
proof (*cases*, *goal-cases*)
case ($1\ d'$)
from *this*(2) $G2(3)$ **have** $u\ c - u\ c' - u\ c < d' + d2$ **using** *add-le-less-mono* **by** *fastforce*
hence $-\ u\ c' < d' + d2$ **by** *simp*
hence $-\ u\ c' < d2 + d'$
by (*metis* (*no-types*) *diff-0-right diff-minus-eq-add minus-add-distrib minus-diff-eq*)
thus *?case* **using** *1* ‹$c \neq c'$› **by** *fastforce*

next
case (*2 d′*)
from *this*(*2*) *G2*(*3*) **have** $u\ c - u\ c' - u\ c < d2 + d'$ **using** *add-strict-mono* **by** *fastforce*
hence $- u\ c' < d2 + d'$ **by** *simp*
thus *?case* **using** *2* ‹$c \neq c'$› **by** *fastforce*
next
case (*3*) **thus** *?case* **by** *auto*
qed
qed
ultimately show *?thesis*
unfolding ∗∗ **by** (*cases*, *auto*)
qed
qed
next
case *True*
with *2* **show** *?thesis* **unfolding** *DBM-reset-def* **by** *auto*
qed
next
case (*3 c′*)
show *?case*
proof (*cases* $c = c'$)
case *False*
hence *F*:$v\ c' \neq v\ c$ **using** *3* **by** *metis*
hence ∗:$M'\ (v\ c')\ 0 = min\ (dbm\text{-}add\ (M\ (v\ c')\ (v\ c))\ (M\ (v\ c)\ 0))\ (M\ (v\ c')\ 0)$
using *F 3* **unfolding** *DBM-reset-def* **by** *simp*
show *?thesis*
proof (*cases* $dbm\text{-}add\ (M\ (v\ c')\ (v\ c))\ (M\ (v\ c)\ 0) < M\ (v\ c')\ 0$)
case *False*
with ∗ **have** $M'\ (v\ c')\ 0 = M\ (v\ c')\ 0$ **by** (*auto split*: *split-min*)
hence *dbm-entry-val u* (*Some c′*) *None* (M' (*v c′*) *0*)
using *3* **unfolding** *DBM-val-bounded-def* **by** *auto*
thus *?thesis* **using** *F* **by** *cases fastforce+*
next
case *True*
with ∗ **have** ∗∗:$M'\ (v\ c')\ 0 = dbm\text{-}add\ (M\ (v\ c')\ (v\ c))\ (M\ (v\ c)\ 0)$ **by** (*auto split*: *split-min*)
from *3* **have** ∗∗∗:*dbm-entry-val u* (*Some c′*) (*Some c*) (*M* (*v c′*) (*v c*))
dbm-entry-val u (*Some c*) *None* (*M* (*v c*) *0*)
unfolding *DBM-val-bounded-def* **by** *auto*
thus *?thesis*
proof −
note ∗∗∗

moreover have *dbm-entry-val* (*u*(*c* := *d*)) (*Some c'*) *None* (*dbm-add* (*Le d1*) (*M* (*v c*) *0*))
if *M* (*v c'*) (*v c*) = *Le d1*
and *dbm-entry-val u* (*Some c*) *None* (*M* (*v c*) *0*)
and *u c'* − *u c* ≤ *d1*
for *d1* :: *'b*
proof −
note *G1* = *that*
from *G1*(*2*) **show** *?thesis*
proof (*cases*, *goal-cases*)
case (*1 d'*)
from *this*(*2*) *G1*(*3*) **have** *u c'* ≤ *d1* + *d'* **using** *ordered-ab-semigroup-add-class.add-mono*
by *fastforce*
thus *?case* **using** *1* ‹*c* ≠ *c'*› **by** *fastforce*
next
case (*2 d'*)
from *this*(*2*) *G1*(*3*) **have** *u c* + *u c'* − *u c* < *d1* + *d'* **using** *add-le-less-mono* **by** *fastforce*
hence *u c'* < *d1* + *d'* **by** *simp*
thus *?case* **using** *2* ‹*c* ≠ *c'*› **by** *fastforce*
next
case (*3*) **thus** *?case* **by** *auto*
qed
qed
moreover have *dbm-entry-val* (*u*(*c* := *d*)) (*Some c'*) *None* (*dbm-add* (*Lt d1*) (*M* (*v c*) *0*))
if *M* (*v c'*) (*v c*) = *Lt d1*
and *dbm-entry-val u* (*Some c*) *None* (*M* (*v c*) *0*)
and *u c'* − *u c* < *d1*
for *d1* :: *'b*
proof −
note *G2* = *that*
from *that*(*2*) **show** *?thesis*
proof (*cases*, *goal-cases*)
case (*1 d'*)
from *this*(*2*) *G2*(*3*) **have** *u c* + *u c'* − *u c* < *d'* + *d1* **using** *add-le-less-mono* **by** *fastforce*
hence *u c'* < *d'* + *d1* **by** *simp*
hence *u c'* < *d1* + *d'*
by (*metis* (*no-types*) *diff-0-right diff-minus-eq-add minus-add-distrib minus-diff-eq*)
thus *?case* **using** *1* ‹*c* ≠ *c'*› **by** *fastforce*
next
case (*2 d'*)

from *this*(*2*) *G2*(*3*) **have** *u c* + *u c'* − *u c* < *d1* + *d'* **using** *add-strict-mono* **by** *fastforce*
hence *u c'* < *d1* + *d'* **by** *simp*
thus *?case* **using** *2* ‹*c* ≠ *c'*› **by** *fastforce*
next
case *3* **thus** *?case* **by** *auto*
qed
qed
ultimately show *?thesis*
unfolding ∗∗ **by** (*cases*, *auto*)
qed
qed
next
case *True*
with *3* **show** *?thesis* **unfolding** *DBM-reset-def* **by** *auto*
qed
next
case (*4 c1 c2*)
show *?case*
proof (*cases c* = *c1*)
case *False*
note *F1* = *this*
show *?thesis*
proof (*cases c* = *c2*)
case *False*
with *F1 4* **have** *F*: *v c* ≠ *v c1 v c* ≠ *v c2 v c1* ≠ *0 v c2* ≠ *0* **by** *force*+
hence ∗:*M'* (*v c1*) (*v c2*) = *min* (*dbm-add* (*M* (*v c1*) (*v c*)) (*M* (*v c*) (*v c2*))) (*M* (*v c1*) (*v c2*))
using *4* **unfolding** *DBM-reset-def* **by** *simp*
show *?thesis*
proof (*cases dbm-add* (*M* (*v c1*) (*v c*)) (*M* (*v c*) (*v c2*)) < *M* (*v c1*) (*v c2*))
case *False*
with ∗ **have** *M'* (*v c1*) (*v c2*) = *M* (*v c1*) (*v c2*) **by** (*auto split*: *split-min*)
hence *dbm-entry-val u* (*Some c1*) (*Some c2*) (*M'* (*v c1*) (*v c2*))
using *4* **unfolding** *DBM-val-bounded-def* **by** *auto*
thus *?thesis* **using** *F* **by** *cases fastforce*+
next
case *True*
with ∗ **have** ∗∗:*M'* (*v c1*) (*v c2*) = *dbm-add* (*M* (*v c1*) (*v c*)) (*M* (*v c*) (*v c2*)) **by** (*auto split*: *split-min*)
from *4* **have** ∗∗∗:*dbm-entry-val u* (*Some c1*) (*Some c*) (*M* (*v c1*) (*v*

```
c))
          dbm-entry-val u (Some c) (Some c2) (M (v c) (v c2)) unfolding
DBM-val-bounded-def by auto
        show ?thesis
        proof -
          note ***
          moreover have dbm-entry-val (u(c := d)) (Some c1) (Some c2)
(dbm-add (Le d1) (M (v c) (v c2)))
            if M (v c1) (v c) = Le d1
              and dbm-entry-val u (Some c) (Some c2) (M (v c) (v c2))
              and u c1 − u c ≤ d1
            for d1 :: 'b
          proof -
            note G1 = that
            from G1(2) show ?thesis
            proof (cases, goal-cases)
              case (1 d')
              from ‹u c − u c2 ≤ d'› ‹u c1 − u c ≤ d1› have u c1 − u c2
≤ d1 + d'
                by (metis (no-types) ab-semigroup-add-class.add-ac(1)
add-le-cancel-right
                                  add-left-mono diff-add-cancel dual-order.refl
dual-order.trans)
              thus ?case using 1(1) ‹c ≠ c1› ‹c ≠ c2› by fastforce
            next
              case (2 d')
              from add-less-le-mono[OF ‹u c − u c2 < d'› ‹u c1 − u c ≤
d1›] have
                − u c2 + u c1 < d' + d1 by simp
              hence u c1 − u c2 < d1 + d' by (simp add: add.commute)
              thus ?case using 2 ‹c ≠ c1› ‹c ≠ c2› by fastforce
            next
              case (3) thus ?case by auto
            qed
          qed
          moreover have dbm-entry-val (u(c := d)) (Some c1) (Some c2)
(dbm-add (Lt d2) (M (v c) (v c2)))
            if M (v c1) (v c) = Lt d2
              and dbm-entry-val u (Some c) (Some c2) (M (v c) (v c2))
              and u c1 − u c < d2
            for d2 :: 'b
          proof -
            note G2 = that
            from G2(2) show ?thesis
```

```
          proof (cases, goal-cases)
            case (1 d′)
              with add-less-le-mono[OF G2(3) this(2)] ‹c ≠ c1› ‹c ≠ c2›
show ?case
                by auto
          next
            case (2 d′)
              with add-strict-mono[OF this(2) G2(3)] ‹c ≠ c1› ‹c ≠ c2›
show ?case
                by (auto simp: add.commute)
          next
            case (3) thus ?case by auto
          qed
        qed
        ultimately show ?thesis
          unfolding ** by (cases, auto)
      qed
    qed
  next
    case True
    with F1 4 have F: v c ≠ v c1 v c1 ≠ 0 v c2 ≠ 0 by force+
    thus ?thesis using 4 True unfolding DBM-reset-def by auto
  qed
next
  case True
  note T1 = this
  show ?thesis
  proof (cases c = c2)
    case False
    with T1 4 have F: v c ≠ v c2 v c1 ≠ 0 v c2 ≠ 0 by force+
    thus ?thesis using 4 True unfolding DBM-reset-def by auto
  next
    case True
    then have *: M′ (v c1) (v c1) = M (v c1) (v c1)
    using T1 4 unfolding DBM-reset-def by auto
     from 4 True T1 have dbm-entry-val u (Some c1) (Some c2) (M (v
c1) (v c2))
    unfolding DBM-val-bounded-def by auto
     then show ?thesis by (cases rule: dbm-entry-val.cases, auto simp: *
True[symmetric] T1)
  qed
 qed
qed
```

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*
using *assms DBM-reset-complete* **by** *metis*

lemma *DBM-reset-diag-preservation*:
  ∀ k≤n. M′ k k ≤ 0 **if** *∀ k≤n. M k k ≤ 0 DBM-reset M n i d M′*
proof *safe*
  fix *k* :: *nat*
  assume *k ≤ n*
  with *that* **show** *M′ k k ≤ 0*
    by (*cases k = i*; *cases k = 0*)
      (*auto simp add*: *DBM-reset-def less*[*symmetric*] *neutral split*: *split-min*)
qed

lemma *FW-diag-preservation*:
  ∀ k≤n. M k k ≤ 0 ⟹ ∀ k≤n. (*FW M n*) *k k ≤ 0*
proof *clarify*
  fix *k* **assume** *A*: *∀ k≤n. M k k ≤ 0 k ≤ n*
  then have *M k k ≤ 0* **by** *auto*
  with *fw-mono*[*of k n k M n*] *A* **show** *FW M n k k ≤ 0* **by** *auto*
qed

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 −
  from *assms*(*1*) **obtain** *i xs* **where** *i ≤ n set xs ⊆ {0..n} len M i i xs <*
Le 0
  unfolding *neutral* **by** *auto*
   with *DBM-reset-neg-cycle-preservation*[*OF assms*(*2*) *this*(*3*)] *assms*(*3*)
obtain *j ys* **where**
    set (*j # ys*) *⊆ {0..n} len M′ j j ys < Le 0*
  by *auto*
  then show *?thesis* **unfolding** *neutral* **by** *force*
qed

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 −
  from *assms*(*5*) **have** $[M]_{v,n} = \{\}$ **unfolding** *DBM-zone-repr-def* **by** *auto*

from *empty-not-cyc-free*[*OF* - *this*] **have** ¬ *cyc-free M n* **using** *assms*(*2*)
by *auto*
from *DBM-reset-not-cyc-free-preservation*[*OF this assms*(*4*,*3*)] **have** ¬ *cyc-free M′ n* **by** *auto*
then obtain *i xs* **where** $i \le n$ *set xs* ⊆ {*0..n*} *len M′ i i xs* < *0* **by** *auto*
from *DBM-val-bounded-neg-cycle*[*OF* - *this assms*(*1*)] **show** *?thesis* **by** *fast*
qed

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 −
note *A* = *assms*
from *A*(*4*) **have** $[FW\ M\ n]_{v,n}$ = {} **unfolding** *DBM-zone-repr-def* **by** *auto*
with *FW-detects-empty-zone*[*OF A*(*1*), *of M*] *A*(*2*)
obtain *i* **where** *i*: $i \le n$ *FW M n i i* < *Le 0* **by** *blast*
with *A*(*3*,*4*) **have** *M′ i i* < *Le 0*
unfolding *DBM-reset-def* **by** (*cases i* = *v c*, *auto split*: *split-min*)
with *fw-mono*[*of i n i M′ n*] *i* **have** *FW M′ n i i* < *Le 0* **by** *auto*
with *FW-detects-empty-zone*[*OF A*(*1*), *of M′*] *A*(*2*) *i*
have $[FW\ M'\ n]_{v,n}$ = {} **by** *auto*
with *FW-zone-equiv*[*OF A*(*1*)] **show** *?thesis* **by** (*auto simp*: *DBM-zone-repr-def*)
qed

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 −
from *assms* **have** $[M]_{v,n}$ = {} **unfolding** *DBM-zone-repr-def* **by** *auto*
with *FW-zone-equiv*[*OF assms*(*1*)] **have**
∀ *u* . ¬ *DBM-val-bounded v u* (*FW M n*) *n*
unfolding *DBM-zone-repr-def* **by** *auto*
from *DBM-reset-complete-empty*[*OF assms*(*1*−*3*) *this*] **show** *?thesis* **by** *auto*
qed

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 $\forall k \le n.\ k > 0 \longrightarrow (\exists c.\ v\ c = k)$ *clock-numbering*$'$ $v\ n\ v\ c \le n$
DBM-reset $M\ n\ (v\ c)\ d\ M'$
shows $[M]_{v,n} = \{\} \longleftrightarrow [M']_{v,n} = \{\}$
proof
assume A: $[M]_{v,n} = \{\}$
hence $\forall\ u\ .\ \neg$ *DBM-val-bounded* $v\ u\ M\ n$ **unfolding** *DBM-zone-repr-def*
by *auto*
hence $\forall\ u\ .\ \neg$ *DBM-val-bounded* $v\ u\ M'\ n$
using *DBM-reset-complete-empty*$'$[*OF assms*(*1*) - *assms*(*3*,*4*)] *assms*(*2*)
by *auto*
thus $[M']_{v,n} = \{\}$ **unfolding** *DBM-zone-repr-def* **by** *auto*
next
assume $[M']_{v,n} = \{\}$
hence $\forall\ u\ .\ \neg$ *DBM-val-bounded* $v\ u\ M'\ n$ **unfolding** *DBM-zone-repr-def*
by *auto*
hence $\forall\ u\ .\ \neg$ *DBM-val-bounded* $v\ u\ M\ n$ **using** *DBM-reset-sound-empty*[*OF*
assms(*2*−*4*)] **by** *auto*
thus $[M]_{v,n} = \{\}$ **unfolding** *DBM-zone-repr-def* **by** *auto*
qed

lemma *DBM-reset-empty*:
assumes $\forall k \le n.\ k > 0 \longrightarrow (\exists c.\ v\ c = k)$ *clock-numbering*$'$ $v\ n\ v\ c \le n$
DBM-reset $(FW\ M\ n)\ n\ (v\ c)\ d\ M'$
shows $[FW\ M\ n]_{v,n} = \{\} \longleftrightarrow [M']_{v,n} = \{\}$
proof
assume A: $[FW\ M\ n]_{v,n} = \{\}$
hence $\forall\ u\ .\ \neg$ *DBM-val-bounded* $v\ u\ (FW\ M\ n)\ n$ **unfolding** *DBM-zone-repr-def*
by *auto*
hence $\forall\ u\ .\ \neg$ *DBM-val-bounded* $v\ u\ M'\ n$
using *DBM-reset-complete-empty*[*of n v M*, *OF assms*(*1*) - *assms*(*4*)]
assms(*2*,*3*) **by** *auto*
thus $[M']_{v,n} = \{\}$ **unfolding** *DBM-zone-repr-def* **by** *auto*
next
assume $[M']_{v,n} = \{\}$
hence $\forall\ u\ .\ \neg$ *DBM-val-bounded* $v\ u\ M'\ n$ **unfolding** *DBM-zone-repr-def*
by *auto*
hence $\forall\ u\ .\ \neg$ *DBM-val-bounded* $v\ u\ (FW\ M\ n)\ n$ **using** *DBM-reset-sound-empty*[*OF*
assms(*2*−)] **by** *auto*
thus $[FW\ M\ n]_{v,n} = \{\}$ **unfolding** *DBM-zone-repr-def* **by** *auto*
qed

lemma *DBM-reset-empty′*:
assumes *canonical M n* $\forall k \leq n.\ k > 0 \longrightarrow (\exists c.\ v\ c = k)$ *clock-numbering′ v n* $v\ c \leq n$
DBM-reset (*FW M n*) *n* (*v c*) *d M′*
shows $[M]_{v,n} = \{\} \longleftrightarrow [M']_{v,n} = \{\}$
using *FW-canonical-id*[*OF assms*(*1*)] *DBM-reset-empty*[*OF assms*(*2*−)] **by** *simp*

lemma *DBM-reset-sound′*:
assumes *clock-numbering′ v n* $v\ c \leq n$ *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 −
from *assms*(*1*) **have**
$\forall c.\ 0 < v\ c$
and $\forall x\ y.\ v\ x \leq n \wedge v\ y \leq n \wedge v\ x = v\ y \longrightarrow x = y$
by *auto*
note *A* = *that assms*(*2*−) *this*
obtain *S-Min-Le* **where** *S-Min-Le*:
S-Min-Le = $\{u\ c' - d \mid c'\ d.\ 0 < v\ c' \wedge v\ c' \leq n \wedge c \neq c' \wedge M\ (v\ c')\ (v\ c) = Le\ d\}$
$\cup \{-d \mid d.\ M\ 0\ (v\ c) = Le\ d\}$ **by** *auto*
obtain *S-Min-Lt* **where** *S-Min-Lt*:
S-Min-Lt = $\{u\ c' - d \mid c'\ d.\ 0 < v\ c' \wedge v\ c' \leq n \wedge c \neq c' \wedge M\ (v\ c')\ (v\ c) = Lt\ d\}$
$\cup \{-d \mid d.\ M\ 0\ (v\ c) = Lt\ d\}$ **by** *auto*
obtain *S-Max-Le* **where** *S-Max-Le*:
S-Max-Le = $\{u\ c' + d \mid c'\ d.\ 0 < v\ c' \wedge v\ c' \leq n \wedge c \neq c' \wedge M\ (v\ c)\ (v\ c') = Le\ d\}$
$\cup \{d \mid d.\ M\ (v\ c)\ 0 = Le\ d\}$ **by** *auto*
obtain *S-Max-Lt* **where** *S-Max-Lt*:
S-Max-Lt = $\{u\ c' + d \mid c'\ d.\ 0 < v\ c' \wedge v\ c' \leq n \wedge c \neq c' \wedge M\ (v\ c)\ (v\ c') = Lt\ d\}$
$\cup \{d \mid d.\ M\ (v\ c)\ 0 = Lt\ d\}$ **by** *auto*

have *finite* $\{c.\ 0 < v\ c \wedge v\ c \leq n\}$ **using** *A*(*6*,*7*)
proof (*induction n*)
case *0*
then have $\{c.\ 0 < v\ c \wedge v\ c \leq 0\} = \{\}$ **by** *auto*
then show *?case* **by** (*metis finite.emptyI*)
next
case (*Suc n*)
then have *finite* $\{c.\ 0 < v\ c \wedge v\ c \leq n\}$ **by** *auto*

moreover have $\{c.\ 0 < v\ c \wedge v\ c \leq Suc\ n\} = \{c.\ 0 < v\ c \wedge v\ c \leq n\}$
$\cup \{c.\ v\ c = Suc\ n\}$ **by** *auto*

moreover have *finite* $\{c.\ v\ c = Suc\ n\}$
proof –
{fix c **assume** $v\ c = Suc\ n$
then have $\{c.\ v\ c = Suc\ n\} = \{c\}$ **using** *Suc.prems(2)* **by** *auto*
}
then show *?thesis* **by** (*cases* $\{c.\ v\ c = Suc\ n\} = \{\}$) *auto*
qed
ultimately show *?case* **by** *auto*
qed
then have $\forall\ f.\ finite\ \{(c,b) \mid c\ b.\ 0 < v\ c \wedge v\ c \leq n \wedge f\ M\ (v\ c) = b\}$
by *auto*
moreover have
$\forall\ f\ K.\ \{(c,K\ d) \mid c\ d.\ 0 < v\ c \wedge v\ c \leq n \wedge f\ M\ (v\ c) = K\ d\}$
$\subseteq \{(c,b) \mid c\ b.\ 0 < v\ c \wedge v\ c \leq n \wedge f\ M\ (v\ c) = b\}$
by *auto*
ultimately have B:
$\forall\ f\ K.\ finite\ \{(c,K\ d) \mid c\ d.\ 0 < v\ c \wedge v\ c \leq n \wedge f\ M\ (v\ c) = K\ d\}$
using *finite-subset* **by** *fast*
have $\forall\ f\ K.\ theLe\ o\ K = id \longrightarrow finite\ \{(c,d) \mid c\ d.\ 0 < v\ c \wedge v\ c \leq n$
$\wedge f\ M\ (v\ c) = K\ d\}$
proof (*safe*, *goal-cases*)
case *prems*: (*1 f K*)
then have $(c,\ d) = (\lambda\ (c,b).\ (c,\ theLe\ b))\ (c,\ K\ d)$ **for** $c :: 'a$ **and** d
by (*simp add*: *pointfree-idE*)
then have
$\{(c,d) \mid c\ d.\ 0 < v\ c \wedge v\ c \leq n \wedge f\ M\ (v\ c) = K\ d\}$
$= (\lambda\ (c,b).\ (c,\ theLe\ b))\ `\ \{(c,K\ d) \mid c\ d.\ 0 < v\ c \wedge v\ c \leq n \wedge f\ M\ (v$
$c) = K\ d\}$
by (*force simp*: *split-beta*)
moreover from B **have**
$finite\ ((\lambda\ (c,b).\ (c,\ theLe\ b))\ `\ \{(c,K\ d) \mid c\ d.\ 0 < v\ c \wedge v\ c \leq n \wedge f$
$M\ (v\ c) = K\ d\})$
by *auto*
ultimately show *?case* **by** *auto*
qed
then have *finI*:
$\bigwedge f\ g\ K.\ theLe\ o\ K = id \Longrightarrow finite\ (g\ `\ \{(c',d) \mid c'\ d.\ 0 < v\ c' \wedge v\ c' \leq$
$n \wedge f\ M\ (v\ c') = K\ d\})$
by *auto*
have *finI1*:
$\bigwedge f\ g\ K.\ theLe\ o\ K = id \Longrightarrow finite\ (g\ `\ \{(c',d) \mid c'\ d.\ 0 < v\ c' \wedge v\ c' \leq$

$n \wedge c \neq c' \wedge f\ M\ (v\ c') = K\ d\})$

proof *goal-cases*

case $(1\ f\ g\ K)$

have

$g\ `\ \{(c',d) \mid c'\ d.\ 0 < v\ c' \wedge v\ c' \leq n \wedge c \neq c' \wedge f\ M\ (v\ c') = K\ d\}$

$\subseteq g\ `\ \{(c',d) \mid c'\ d.\ 0 < v\ c' \wedge v\ c' \leq n \wedge f\ M\ (v\ c') = K\ d\}$

by *auto*

from *finite-subset*[*OF this finI*[*OF 1*, *of g f*]] **show** *?case* **.**

qed

have $\forall\ f.$ *finite* $\{b.\ f\ M\ (v\ c) = b\}$ **by** *auto*

moreover have $\forall\ f\ K.\ \{K\ d \mid d.\ f\ M\ (v\ c) = K\ d\} \subseteq \{b.\ f\ M\ (v\ c) = b\}$ **by** *auto*

ultimately have B: $\forall\ f\ K.$ *finite* $\{K\ d \mid d.\ f\ M\ (v\ c) = K\ d\}$ **using** *finite-subset* **by** *fast*

have $\forall\ f\ K.$ *theLe o* $K = id \longrightarrow$ *finite* $\{d \mid d.\ f\ M\ (v\ c) = K\ d\}$

proof (*safe*, *goal-cases*)

case *prems*: $(1\ f\ K)$

then have $(c,\ d) = (\lambda\ (c,b).\ (c,$ *theLe* $b))\ (c,\ K\ d)$ **for** $c :: 'a$ **and** d

by (*simp add*: *pointfree-idE*)

then have

$\{d \mid d.\ f\ M\ (v\ c) = K\ d\}$

$= (\lambda\ b.$ *theLe* $b)\ `\ \{K\ d \mid d.\ f\ M\ (v\ c) = K\ d\}$

by (*force simp*: *split-beta*)

moreover from B **have**

finite $((\lambda b.$ *theLe* $b)\ `\ \{K\ d \mid d.\ f\ M\ (v\ c) = K\ d\})$

by *auto*

ultimately show *?case* **by** *auto*

qed

then have C: $\forall\ f\ g\ K.$ *theLe o* $K = id \longrightarrow$ *finite* $(g\ `\ \{d \mid d.\ f\ M\ (v\ c) = K\ d\})$ **by** *auto*

have *finI2*: $\bigwedge f\ g\ K.$ *theLe o* $K = id \Longrightarrow$ *finite* $(\{g\ d \mid d.\ f\ M\ (v\ c) = K\ d\})$

proof *goal-cases*

case $(1\ f\ g\ K)$

have $\{g\ d \mid d.\ f\ M\ (v\ c) = K\ d\} = g\ `\ \{d \mid d.\ f\ M\ (v\ c) = K\ d\}$ **by** *auto*

with $C\ 1$ **show** *?case* **by** *auto*

qed

{ fix $K :: 'b \Rightarrow 'b$ *DBMEntry* **assume** A: *theLe o* $K = id$

then have

finite $((\lambda(c,d).\ u\ c - d)\ `\ \{(c',d) \mid c'\ d.\ 0 < v\ c' \wedge v\ c' \leq n \wedge c \neq c' \wedge M\ (v\ c')\ (v\ c) = K\ d\})$

by (*intro finI1*, *auto*)

moreover have
$\{u\ c' - d\ |c'\ d.\ 0 < v\ c' \wedge v\ c' \leq n \wedge c \neq c' \wedge M\ (v\ c')\ (v\ c) = K\ d\}$
$= ((\lambda(c,d).\ u\ c - d)\ `\ \{(c',d)\ |\ c'\ d.\ 0 < v\ c' \wedge v\ c' \leq n \wedge c \neq c' \wedge M\ (v\ c')\ (v\ c) = K\ d\})$
by *auto*
ultimately have *finite* $\{u\ c' - d\ |c'\ d.\ 0 < v\ c' \wedge v\ c' \leq n \wedge c \neq c' \wedge M\ (v\ c')\ (v\ c) = K\ d\}$
by *auto*
moreover have *finite* $\{- d\ |d.\ M\ 0\ (v\ c) = K\ d\}$ **using** A **by** (*intro finI2*, *auto*)
ultimately have
finite $(\{u\ c' - d\ |c'\ d.\ 0 < v\ c' \wedge v\ c' \leq n \wedge c \neq c' \wedge M\ (v\ c')\ (v\ c) = K\ d\}$
$\cup \{- d\ |d.\ M\ 0\ (v\ c) = K\ d\})$
by (*auto simp*: *S-Min-Le*)
} **note** *fin1* = *this*
have *fin-min-le*: *finite S-Min-Le* **unfolding** *S-Min-Le* **by** (*rule fin1*, *auto*)
have *fin-min-lt*: *finite S-Min-Lt* **unfolding** *S-Min-Lt* **by** (*rule fin1*, *auto*)

{ **fix** K :: $'b \Rightarrow\ 'b$ *DBMEntry* **assume** A: *theLe o K = id*
then have *finite* $((\lambda(c,d).\ u\ c + d)\ `\ \{(c',d)\ |\ c'\ d.\ 0 < v\ c' \wedge v\ c' \leq n \wedge c \neq c' \wedge M\ (v\ c)\ (v\ c') = K\ d\})$
by (*intro finI1*, *auto*)
moreover have
$\{u\ c' + d\ |c'\ d.\ 0 < v\ c' \wedge v\ c' \leq n \wedge c \neq c' \wedge M\ (v\ c)\ (v\ c') = K\ d\}$
$= ((\lambda(c,d).\ u\ c + d)\ `\ \{(c',d)\ |\ c'\ d.\ 0 < v\ c' \wedge v\ c' \leq n \wedge c \neq c' \wedge M\ (v\ c)\ (v\ c') = K\ d\})$
by *auto*
ultimately have *finite* $\{u\ c' + d\ |c'\ d.\ 0 < v\ c' \wedge v\ c' \leq n \wedge c \neq c' \wedge M\ (v\ c)\ (v\ c') = K\ d\}$
by *auto*
moreover have *finite* $\{d\ |d.\ M\ (v\ c)\ 0 = K\ d\}$ **using** A **by** (*intro finI2*, *auto*)
ultimately have
finite $(\{u\ c' + d\ |c'\ d.\ 0 < v\ c' \wedge v\ c' \leq n \wedge c \neq c' \wedge M\ (v\ c)\ (v\ c') = K\ d\}$
$\cup \{d\ |d.\ M\ (v\ c)\ 0 = K\ d\})$
by (*auto simp*: *S-Min-Le*)
} **note** *fin2* = *this*
have *fin-max-le*: *finite S-Max-Le* **unfolding** *S-Max-Le* **by** (*rule fin2*, *auto*)
have *fin-max-lt*: *finite S-Max-Lt* **unfolding** *S-Max-Lt* **by** (*rule fin2*, *auto*)

{ **fix** $l\ r$ **assume** $l \in$ *S-Min-Le* $r \in$ *S-Max-Le*
then have $l \leq r$

unfolding *S-Min-Le S-Max-Le*
proof (*safe*, *goal-cases*)
case (*1 c1 d1 c2 d2*)
with *A* **have**
dbm-entry-val u (*Some c1*) (*Some c2*) (M' (*v c1*) (*v c2*))
unfolding *DBM-val-bounded-def* **by** *presburger*
moreover have
M' (*v c1*) (*v c2*) = *min* (*dbm-add* (*M* (*v c1*) (*v c*)) (*M* (*v c*) (*v c2*))) (*M* (*v c1*) (*v c2*))
using *A*(*3*,*7*) *1* **unfolding** *DBM-reset-def* **by** *metis*
ultimately have
dbm-entry-val u (*Some c1*) (*Some c2*) (*dbm-add* (*M* (*v c1*) (*v c*)) (*M* (*v c*) (*v c2*)))
using *dbm-entry-dbm-min′* **by** *auto*
with *1* **have** $u\ c1 - u\ c2 \leq d1 + d2$ **by** *auto*
thus *?case*
by (*metis* (*no-types*) *add-diff-cancel-left diff-0-right diff-add-cancel diff-eq-diff-less-eq*)
next
case (*2* c' *d*)
with *A* **have**
($\forall i \leq n.\ i \neq v\ c \wedge i > 0 \longrightarrow M'\ i\ 0 = min\ (dbm\text{-}add\ (M\ i\ (v\ c))\ (M\ (v\ c)\ 0))\ (M\ i\ 0)$)
$v\ c' \neq v\ c$
unfolding *DBM-reset-def* **by** *auto*
hence (M' (*v* c') *0* = *min* (*dbm-add* (*M* (*v* c') (*v c*)) (*M* (*v c*) *0*)) (*M* (*v* c') *0*))
using *2* **by** *blast*
moreover from *A 2* **have** *dbm-entry-val u* (*Some* c') *None* (M' (*v* c') *0*)
unfolding *DBM-val-bounded-def* **by** *presburger*
ultimately have *dbm-entry-val u* (*Some* c') *None* (*dbm-add* (*M* (*v* c') (*v c*)) (*M* (*v c*) *0*))
using *dbm-entry-dbm-min3′* **by** *fastforce*
with *2* **have** $u\ c' \leq d + r$ **by** *auto*
thus *?case* **by** (*metis add-diff-cancel-left add-le-cancel-right diff-0-right diff-add-cancel*)
next
case (*3 d* c' d')
with *A* **have**
($\forall i \leq n.\ i \neq v\ c \wedge i > 0 \longrightarrow M'\ 0\ i = min\ (dbm\text{-}add\ (M\ 0\ (v\ c))\ (M\ (v\ c)\ i))\ (M\ 0\ i)$)
$v\ c' \neq v\ c$
unfolding *DBM-reset-def* **by** *auto*

hence (M' *0* (v c') = *min* (*dbm-add* (M *0* (v c)) (M (v c) (v c'))) (M *0* (v c')))
using *3* **by** *blast*
moreover from A *3* **have** *dbm-entry-val u None* (*Some* c') (M' *0* (v c'))
unfolding *DBM-val-bounded-def* **by** *presburger*
ultimately have *dbm-entry-val u None* (*Some* c') (*dbm-add* (M *0* (v c)) (M (v c) (v c')))
using *dbm-entry-dbm-min2'* **by** *fastforce*
with *3* **have** $-u\ c' \leq d + d'$ **by** *auto*
thus *?case*
by (*metis add-uminus-conv-diff diff-le-eq minus-add-distrib minus-le-iff*)
next
case (*4* d)

Here is the reason we need the assumption that the zone was not empty before the reset. We cannot deduce anything from the current value of c itself because we reset it. We can only ensure that we can reset the value of c by using the value from the alternative assignment. This case is only relevant if the tightest bounds for d were given by its original lower and upper bounds. If they would overlap, the original zone would be empty.

from A(*2*,*5*) **have**
dbm-entry-val u'' *None* (*Some* c) (M *0* (v c))
dbm-entry-val u'' (*Some* c) *None* (M (v c) *0*)
unfolding *DBM-val-bounded-def* **by** *auto*
with *4* **have** $-\ u''\ c \leq d$ $u''\ c \leq r$ **by** *auto*
thus *?case* **by** (*metis minus-le-iff order.trans*)
qed
} **note** $EE = this$
{ **fix** l r **assume** $l \in$ *S-Min-Le* $r \in$ *S-Max-Lt*
then have $l < r$
unfolding *S-Min-Le S-Max-Lt*
proof (*safe*, *goal-cases*)
case (*1 c1 d1 c2 d2*)
with A **have** *dbm-entry-val u* (*Some c1*) (*Some c2*) (M' (v *c1*) (v *c2*))
unfolding *DBM-val-bounded-def* **by** *presburger*
moreover have M' (v *c1*) (v *c2*) = *min* (*dbm-add* (M (v *c1*) (v c)) (M (v c) (v *c2*))) (M (v *c1*) (v *c2*))
using A(*3*,*7*) *1* **unfolding** *DBM-reset-def* **by** *metis*
ultimately have *dbm-entry-val u* (*Some c1*) (*Some c2*) (*dbm-add* (M (v *c1*) (v c)) (M (v c) (v *c2*)))
using *dbm-entry-dbm-min'* **by** *fastforce*
with *1* **have** $u\ c1 - u\ c2 < d1 + d2$ **by** *auto*
then show *?case* **by** (*metis add.assoc add.commute diff-less-eq*)

next
case (*2* c' *d*)
with *A* **have**
($\forall i \le n.\ i \neq v\ c \wedge i > 0 \longrightarrow M'\ i\ 0 = min\ (dbm\text{-}add\ (M\ i\ (v\ c))\ (M\ (v\ c)\ 0))\ (M\ i\ 0)$)
$v\ c' \neq v\ c$
unfolding *DBM-reset-def* **by** *auto*
hence ($M'\ (v\ c')\ 0 = min\ (dbm\text{-}add\ (M\ (v\ c')\ (v\ c))\ (M\ (v\ c)\ 0))\ (M\ (v\ c')\ 0)$)
using *2* **by** *blast*
moreover from *A 2* **have** *dbm-entry-val u* (*Some* c') *None* ($M'\ (v\ c')\ 0$)
unfolding *DBM-val-bounded-def* **by** *presburger*
ultimately have *dbm-entry-val u* (*Some* c') *None* ($dbm\text{-}add\ (M\ (v\ c')\ (v\ c))\ (M\ (v\ c)\ 0)$)
using *dbm-entry-dbm-min3'* **by** *fastforce*
with *2* **have** $u\ c' < d + r$ **by** *auto*
thus *?case* **by** (*metis add-less-imp-less-right diff-add-cancel gt-swap*)
next
case (*3* *d* c' *da*)
with *A* **have**
($\forall i \le n.\ i \neq v\ c \wedge i > 0 \longrightarrow M'\ 0\ i = min\ (dbm\text{-}add\ (M\ 0\ (v\ c))\ (M\ (v\ c)\ i))\ (M\ 0\ i)$)
$v\ c' \neq v\ c$
unfolding *DBM-reset-def* **by** *auto*
hence ($M'\ 0\ (v\ c') = min\ (dbm\text{-}add\ (M\ 0\ (v\ c))\ (M\ (v\ c)\ (v\ c')))\ (M\ 0\ (v\ c'))$)
using *3* **by** *blast*
moreover from *A 3* **have** *dbm-entry-val u None* (*Some* c') ($M'\ 0\ (v\ c')$)
unfolding *DBM-val-bounded-def* **by** *presburger*
ultimately have *dbm-entry-val u None* (*Some* c') ($dbm\text{-}add\ (M\ 0\ (v\ c))\ (M\ (v\ c)\ (v\ c'))$)
using *dbm-entry-dbm-min2'* **by** *fastforce*
with *3* **have** $-u\ c' < d + da$ **by** *auto*
thus *?case* **by** (*metis add.commute diff-less-eq uminus-add-conv-diff*)
next
case (*4* *d*)
from *A*(*2*,*5*) **have**
dbm-entry-val u'' *None* (*Some c*) ($M\ 0\ (v\ c)$)
dbm-entry-val u'' (*Some c*) *None* ($M\ (v\ c)\ 0$)
unfolding *DBM-val-bounded-def* **by** *auto*
with *4* **have** $-\ u''\ c \le d\ \ u''\ c < r$ **by** *auto*
thus *?case* **by** (*metis minus-le-iff neq-iff not-le order.strict-trans*)

qed
} **note** *EL* = *this*
{ **fix** *l r* **assume** *l* ∈ *S-Min-Lt r* ∈ *S-Max-Le*
then have *l* < *r*
unfolding *S-Min-Lt S-Max-Le*
proof (*safe*, *goal-cases*)
case (*1 c1 d1 c2 d2*)
with *A* **have** *dbm-entry-val u* (*Some c1*) (*Some c2*) (*M′* (*v c1*) (*v c2*))
unfolding *DBM-val-bounded-def* **by** *presburger*
moreover have *M′* (*v c1*) (*v c2*) = *min* (*dbm-add* (*M* (*v c1*) (*v c*)) (*M* (*v c*) (*v c2*))) (*M* (*v c1*) (*v c2*))
using *A*(*3*,*7*) *1* **unfolding** *DBM-reset-def* **by** *metis*
ultimately have *dbm-entry-val u* (*Some c1*) (*Some c2*) (*dbm-add* (*M* (*v c1*) (*v c*)) (*M* (*v c*) (*v c2*)))
using *dbm-entry-dbm-min′* **by** *fastforce*
with *1* **have** *u c1* − *u c2* < *d1* + *d2* **by** *auto*
thus *?case* **by** (*metis add.assoc add.commute diff-less-eq*)
next
case (*2 c′ d*)
with *A* **have**
(∀ *i*≤*n*. *i* ≠ *v c* ∧ *i* > *0* ⟶ *M′ i 0* = *min* (*dbm-add* (*M i* (*v c*)) (*M* (*v c*) *0*)) (*M i 0*))
v c′ ≠ *v c*
unfolding *DBM-reset-def* **by** *auto*
hence (*M′* (*v c′*) *0* = *min* (*dbm-add* (*M* (*v c′*) (*v c*)) (*M* (*v c*) *0*)) (*M* (*v c′*) *0*))
using *2* **by** *blast*
moreover from *A 2* **have** *dbm-entry-val u* (*Some c′*) *None* (*M′* (*v c′*) *0*)
unfolding *DBM-val-bounded-def* **by** *presburger*
ultimately have *dbm-entry-val u* (*Some c′*) *None* (*dbm-add* (*M* (*v c′*) (*v c*)) (*M* (*v c*) *0*))
using *dbm-entry-dbm-min3′* **by** *fastforce*
with *2* **have** *u c′* < *d* + *r* **by** *auto*
thus *?case* **by** (*metis add-less-imp-less-right diff-add-cancel gt-swap*)
next
case (*3 d c′ da*)
with *A* **have**
(∀ *i*≤*n*. *i* ≠ *v c* ∧ *i* > *0* ⟶ *M′ 0 i* = *min* (*dbm-add* (*M 0* (*v c*)) (*M* (*v c*) *i*)) (*M 0 i*))
v c′ ≠ *v c*
unfolding *DBM-reset-def* **by** *auto*
hence (*M′ 0* (*v c′*) = *min* (*dbm-add* (*M 0* (*v c*)) (*M* (*v c*) (*v c′*))) (*M 0* (*v c′*)))

using *3* **by** *blast*
moreover from *A 3* **have** *dbm-entry-val u None* (*Some c′*) (*M′ 0* (*v c′*))
unfolding *DBM-val-bounded-def* **by** *presburger*
ultimately have *dbm-entry-val u None* (*Some c′*) (*dbm-add* (*M 0* (*v c*)) (*M* (*v c*) (*v c′*)))
using *dbm-entry-dbm-min2′* **by** *fastforce*
with *3* **have** $-u\ c' < d + da$ **by** *auto*
thus *?case* **by** (*metis add.commute diff-less-eq uminus-add-conv-diff*)
next
case (*4 d*)
from *A*(*2*,*5*) **have**
dbm-entry-val u′′ None (*Some c*) (*M 0* (*v c*))
dbm-entry-val u′′ (*Some c*) *None* (*M* (*v c*) *0*)
unfolding *DBM-val-bounded-def* **by** *auto*
with *4* **have** $-\ u''\ c < d\ u''\ c \le r$ **by** *auto*
thus *?case* **by** (*meson less-le-trans minus-less-iff*)
qed
} **note** *LE* = *this*
{ **fix** *l r* **assume** $l \in$ *S-Min-Lt* $r \in$ *S-Max-Lt*
then have $l < r$
unfolding *S-Min-Lt S-Max-Lt*
proof (*safe*, *goal-cases*)
case (*1 c1 d1 c2 d2*)
with *A* **have** *dbm-entry-val u* (*Some c1*) (*Some c2*) (*M′* (*v c1*) (*v c2*))
unfolding *DBM-val-bounded-def* **by** *presburger*
moreover have *M′* (*v c1*) (*v c2*) = *min* (*dbm-add* (*M* (*v c1*) (*v c*)) (*M* (*v c*) (*v c2*))) (*M* (*v c1*) (*v c2*))
using *A*(*3*,*7*) *1* **unfolding** *DBM-reset-def* **by** *metis*
ultimately have *dbm-entry-val u* (*Some c1*) (*Some c2*) (*dbm-add* (*M* (*v c1*) (*v c*)) (*M* (*v c*) (*v c2*)))
using *dbm-entry-dbm-min′* **by** *fastforce*
with *1* **have** $u\ c1 - u\ c2 < d1 + d2$ **by** *auto*
then show *?case* **by** (*metis add.assoc add.commute diff-less-eq*)
next
case (*2 c′ d*)
with *A* **have**
($\forall i \le n.\ i \neq v\ c \land i > 0 \longrightarrow M'\ i\ 0 = min$ (*dbm-add* (*M i* (*v c*)) (*M* (*v c*) *0*)) (*M i 0*))
$v\ c' \neq v\ c$
unfolding *DBM-reset-def* **by** *auto*
hence (*M′* (*v c′*) *0* = *min* (*dbm-add* (*M* (*v c′*) (*v c*)) (*M* (*v c*) *0*)) (*M* (*v c′*) *0*))
using *2* **by** *blast*

moreover from *A 2* **have** *dbm-entry-val u* (*Some c′*) *None* (*M′* (*v c′*) *0*)
unfolding *DBM-val-bounded-def* **by** *presburger*
ultimately have *dbm-entry-val u* (*Some c′*) *None* (*dbm-add* (*M* (*v c′*) (*v c*)) (*M* (*v c*) *0*))
using *dbm-entry-dbm-min3′* **by** *fastforce*
with *2* **have** $u\ c' < d + r$ **by** *auto*
thus *?case* **by** (*metis add-less-imp-less-right diff-add-cancel gt-swap*)
next
case (*3 d c′ da*)
with *A* **have**
(∀ *i*≤*n*. $i \neq v\ c \wedge i > 0 \longrightarrow$ *M′ 0 i* = *min* (*dbm-add* (*M 0* (*v c*)) (*M* (*v c*) *i*)) (*M 0 i*))
$v\ c' \neq v\ c$
unfolding *DBM-reset-def* **by** *auto*
hence (*M′ 0* (*v c′*) = *min* (*dbm-add* (*M 0* (*v c*)) (*M* (*v c*) (*v c′*))) (*M 0* (*v c′*)))
using *3* **by** *blast*
moreover from *A 3* **have** *dbm-entry-val u None* (*Some c′*) (*M′ 0* (*v c′*))
unfolding *DBM-val-bounded-def* **by** *presburger*
ultimately have *dbm-entry-val u None* (*Some c′*) (*dbm-add* (*M 0* (*v c*)) (*M* (*v c*) (*v c′*)))
using *dbm-entry-dbm-min2′* **by** *fastforce*
with *3* **have** $-u\ c' < d + da$ **by** *auto*
thus *?case* **by** (*metis ab-group-add-class.ab-diff-conv-add-uminus add.commute diff-less-eq*)
next
case (*4 d*)
from *A*(*2*,*5*) **have**
dbm-entry-val u″ None (*Some c*) (*M 0* (*v c*))
dbm-entry-val u″ (*Some c*) *None* (*M* (*v c*) *0*)
unfolding *DBM-val-bounded-def* **by** *auto*
with *4* **have** $-\ u''\ c \le d\ u''\ c < r$ **by** *auto*
thus *?case* **by** (*metis minus-le-iff neq-iff not-le order.strict-trans*)
qed
} **note** *LL* = *this*

obtain *d′* **where** *d′*:
∀ *t* ∈ *S-Min-Le*. $d' \ge t$ ∀ *t* ∈ *S-Min-Lt*. $d' > t$
∀ *t* ∈ *S-Max-Le*. $d' \le t$ ∀ *t* ∈ *S-Max-Lt*. $d' < t$
proof −
assume *m*:
⋀*d′*. ⟦∀ *t*∈*S-Min-Le*. $t \le d'$; ∀ *t*∈*S-Min-Lt*. $t < d'$; ∀ *t*∈*S-Max-Le*. $d' \le$

```
t; ∀ t∈S-Max-Lt. d′ < t⟧
        ⟹ thesis
    let ?min-le = Max S-Min-Le
    let ?min-lt = Max S-Min-Lt
    let ?max-le = Min S-Max-Le
    let ?max-lt = Min S-Max-Lt

    show thesis
    proof (cases S-Min-Le = {} ∧ S-Min-Lt = {})
      case True
      note T = this
      show thesis
      proof (cases S-Max-Le = {} ∧ S-Max-Lt = {})
        case True
        let ?d′ = 0 :: ′t :: time
        show thesis using True T by (intro m[of ?d′]) auto
      next
        case False
        let ?d =
          if S-Max-Le ≠ {}
           then if S-Max-Lt ≠ {} then min ?max-lt ?max-le else ?max-le
           else ?max-lt
        obtain a :: ′b where a: a < 0 using non-trivial-neg by auto
        let ?d′ = min 0 (?d + a)
        { fix x assume x ∈ S-Max-Le
          with fin-max-le a have min 0 (Min S-Max-Le + a) ≤ x
           by (metis Min.boundedE add-le-same-cancel1 empty-iff less-imp-le
min.coboundedI2)
          then have min 0 (Min S-Max-Le + a) ≤ x by auto
        } note 1 = this
        { fix x assume x: x ∈ S-Max-Lt
         have min 0 (min (Min S-Max-Lt) (Min S-Max-Le) + a) < ?max-lt
       by (meson a add-less-same-cancel1 min.cobounded1 min.strict-coboundedI2
order.strict-trans2)
          also from fin-max-lt x have ... ≤ x by auto
         finally have min 0 (min (Min S-Max-Lt) (Min S-Max-Le) + a) <
x .
        } note 2 = this
        { fix x assume x: x ∈ S-Max-Le
         have min 0 (min (Min S-Max-Lt) (Min S-Max-Le) + a) ≤ ?max-le
          by (metis le-add-same-cancel1 linear not-le a min-le-iff-disj)
          also from fin-max-le x have ... ≤ x by auto
         finally have min 0 (min (Min S-Max-Lt) (Min S-Max-Le) + a) ≤
x .
```

```
      } note 3 = this
      show thesis using False T a 1 2 3
        by (intro m[of ?d'], auto)
       (metis Min.coboundedI add-less-same-cancel1 fin-max-lt min.boundedE
min.orderE
             not-less)
    qed
  next
    case False
    note F = this
    show thesis
    proof (cases S-Max-Le = {} ∧ S-Max-Lt = {})
      case True
      let ?l =
        if S-Min-Le ≠ {}
         then if S-Min-Lt ≠ {} then max ?min-lt ?min-le else ?min-le
         else ?min-lt
      obtain a :: 'b where a < 0 using non-trivial-neg by blast
      then have a: −a > 0 using non-trivial-neg by simp
      then obtain a :: 'b where a: a > 0 by blast
      let ?d' = ?l + a
      {
        fix x assume x: x ∈ S-Min-Le
        then have x ≤ max ?min-lt ?min-le x ≤ ?min-le using fin-min-le
by (simp add: max.coboundedI2)+
         then have x ≤ max ?min-lt ?min-le + a x ≤ ?min-le + a using
a by (simp add: add-increasing2)+
      } note 1 = this
      {
        fix x assume x: x ∈ S-Min-Lt
        then have x ≤ max ?min-lt ?min-le x ≤ ?min-lt using fin-min-lt
by (simp add: max.coboundedI1)+
         then have x < ?d' using a x by (auto simp add: add.commute
add-strict-increasing)
      } note 2 = this
      show thesis using True F a 1 2 by ((intro m[of ?d']), auto)
    next
      case False
      let ?r =
        if S-Max-Le ≠ {}
         then if S-Max-Lt ≠ {} then min ?max-lt ?max-le else ?max-le
         else ?max-lt
      let ?l =
        if S-Min-Le ≠ {}
```

then if S-Min-Lt $\neq$ {} *then max ?min-lt ?min-le else ?min-le*
else ?min-lt
have *1*: $x \leq$ *max ?min-lt ?min-le* $x \leq$ *?min-le* **if** $x \in$ *S-Min-Le* **for** x
by (*simp add*: *max.coboundedI2 that fin-min-le*)+
{
fix *x y* **assume** *x*: $x \in$ *S-Max-Le* $y \in$ *S-Min-Lt*
then have *S-Min-Lt* $\neq$ {} **by** *auto*
from *LE*[*OF Max-in*[*OF fin-min-lt*], *OF this*, *OF x*(*1*)] **have** *?min-lt* $\leq x$ **by** *auto*
} **note** *3* = *this*
{
fix *x y* **assume** *x*: $x \in$ *S-Max-Le* $y \in$ *S-Min-Le*
with *EE*[*OF Max-in*[*OF fin-min-le*], *OF* - *x*(*1*)] **have** *?min-le* $\leq x$ **by** *auto*
} **note** *4* = *this*
{
fix *x y* **assume** *x*: $x \in$ *S-Max-Lt* $y \in$ *S-Min-Lt*
then have *S-Min-Lt* $\neq$ {} **by** *auto*
from *LL*[*OF Max-in*[*OF fin-min-lt*], *OF this*, *OF x*(*1*)] **have** *?min-lt* $< x$ **by** *auto*
} **note** *5* = *this*
{
fix *x y* **assume** *x*: $x \in$ *S-Max-Lt* $y \in$ *S-Min-Le*
then have *S-Min-Le* $\neq$ {} **by** *auto*
from *EL*[*OF Max-in*[*OF fin-min-le*], *OF this*, *OF x*(*1*)] **have** *?min-le* $< x$ **by** *auto*
} **note** *6* = *this*

show *thesis*
proof (*cases ?l* < *?r*)
case *False*
then have $*$: *S-Max-Le* $\neq$ {}
proof (*safe*, *goal-cases*)
case *1*
with ‹¬ (*S-Max-Le* = {} ∧ *S-Max-Lt* = {})› **obtain** *y* **where** *y*:$y \in$ *S-Max-Lt* **by** *auto*
note *1* = *1 this*
{ **fix** *x y* **assume** *A*: $x \in$ *S-Min-Le* $y \in$ *S-Max-Lt*
with *EL*[*OF Max-in*[*OF fin-min-le*] *Min-in*[*OF fin-max-lt*]]
have *Max S-Min-Le* < *Min S-Max-Lt* **by** *auto*
} **note** $**$ = *this*
{ **fix** *x y* **assume** *A*: $x \in$ *S-Min-Lt* $y \in$ *S-Max-Lt*
with *LL*[*OF Max-in*[*OF fin-min-lt*] *Min-in*[*OF fin-max-lt*]]
have *Max S-Min-Lt* < *Min S-Max-Lt* **by** *auto*

} **note** ∗∗∗ = *this*
show *?case*
proof (*cases S-Min-Le* ≠ {})
case *True*
note *T* = *this*
show *?thesis*
proof (*cases S-Min-Lt* ≠ {})
case *True*
then show *False* **using** *1 T True* ∗∗ ∗∗∗ **by** *auto*
next
case *False* **with** *1 T* ∗∗ **show** *False* **by** *auto*
qed
next
case *False*
with *1 False* ∗∗∗ ‹¬ (*S-Min-Le* = {} ∧ *S-Min-Lt* = {})› **show** *?thesis* **by** *auto*
qed
qed
{ **fix** *x y* **assume** *A*: *x* ∈ *S-Min-Lt y* ∈ *S-Max-Lt*
with *LL*[*OF Max-in*[*OF fin-min-lt*] *Min-in*[*OF fin-max-lt*]]
have *Max S-Min-Lt* < *Min S-Max-Lt* **by** *auto*
} **note** ∗∗∗ = *this*
{ **fix** *x y* **assume** *A*: *x* ∈ *S-Min-Lt y* ∈ *S-Max-Le*
with *LE*[*OF Max-in*[*OF fin-min-lt*] *Min-in*[*OF fin-max-le*]]
have *Max S-Min-Lt* < *Min S-Max-Le* **by** *auto*
} **note** ∗∗∗∗ = *this*
from *F False* **have** ∗∗: *S-Min-Le* ≠ {}
proof (*safe*, *goal-cases*)
case *1*
show *?case*
proof (*cases S-Max-Le* ≠ {})
case *True*
note *T* = *this*
show *?thesis*
proof (*cases S-Max-Lt* ≠ {})
case *True*
then show *?thesis* **using** *1 T True* ∗∗∗∗ ∗∗∗ **by** *auto*
next
case *False* **with** *1 T* ∗∗∗∗ **show** *?thesis* **by** *auto*
qed
next
case *False*
with *1 False* ∗∗∗ ‹¬ (*S-Max-Le* = {} ∧ *S-Max-Lt* = {})› **show** *?thesis* **by** *auto*

qed
qed
{
fix *x* **assume** *x*: $x \in$ *S-Min-Lt*
then have $x \leq$ *?min-lt* **using** *fin-min-lt* **by** (*simp add*: *max.coboundedI2*)
also have *?min-lt* < *?min-le*
proof (*rule ccontr*, *goal-cases*)
case *1*
with *x* ** **have** *1*: *?l* = *?min-lt* **by** (*auto simp*: *max.absorb1*)
have *2*: *?min-lt* < *?max-le* **using** * ****[*OF x*] **by** *auto*
show *False*
proof (*cases S-Max-Lt* = {})
case *False*
then have *?min-lt* < *?max-lt* **using** * ***[*OF x*] **by** *auto*
with *1 2* **have** *?l* < *?r* **by** *auto*
with ‹¬ *?l* < *?r*› **show** *False* **by** *auto*
next
case *True*
with *1 2* **have** *?l* < *?r* **by** *auto*
with ‹¬ *?l* < *?r*› **show** *False* **by** *auto*
qed
qed
finally have *x* < *max ?min-lt ?min-le* **by** (*simp add*: *max.strict-coboundedI2*)
} **note** *2* = *this*
show *thesis* **using** *F False 1 2 3 4 5 6* * ** **by** ((*intro m*[*of ?l*]), *auto*)
next
case *True*
then obtain *d* **where** *d*: *?l* < *d d* < *?r* **using** *dense* **by** *auto*
let *?d′* = *d*
{
fix *t* **assume** $t \in$ *S-Min-Le*
then have $t \leq$ *?l* **using** *1* **by** *auto*
with *d* **have** $t \leq d$ **by** *auto*
}
moreover {
fix *t* **assume** *t*: $t \in$ *S-Min-Lt*
then have $t \leq$ *max ?min-lt ?min-le* **using** *fin-min-lt* **by** (*simp add*: *max.coboundedI1*)
with *t* **have** $t \leq$ *?l* **using** *fin-min-lt* **by** *auto*
with *d* **have** $t < d$ **by** *auto*
}
moreover {

fix *t* **assume** *t*: *t* ∈ *S-Max-Le*
then have *min ?max-lt ?max-le* ≤ *t* **using** *fin-max-le* **by** (*simp add*: *min.coboundedI2*)
then have *?r* ≤ *t* **using** *fin-max-le t* **by** *auto*
with *d* **have** *d* ≤ *t* **by** *auto*
then have *d* ≤ *t* **by** (*simp add*: *min.coboundedI2*)
}
moreover {
fix *t* **assume** *t*: *t* ∈ *S-Max-Lt*
then have *min ?max-lt ?max-le* ≤ *t* **using** *fin-max-lt* **by** (*simp add*: *min.coboundedI1*)
then have *?r* ≤ *t* **using** *fin-max-lt t* **by** *auto*
with *d* **have** *d* < *t* **by** *auto*
then have *d* < *t* **by** (*simp add*: *min.strict-coboundedI2*)
}
ultimately show *thesis* **by** ((*intro m*[*of ?d'*]), *auto*)
qed
qed
qed
qed
have *DBM-val-bounded v* (*u*(*c* := *d'*)) *M n* **unfolding** *DBM-val-bounded-def*
proof (*safe*, *goal-cases*)
case *1*
with *A* **show** *?case* **unfolding** *DBM-reset-def DBM-val-bounded-def* **by** *auto*
next
case (*2 c'*)
show *?case*
proof (*cases c* = *c'*)
case *False*
with *A*(*2*,*7*) **have** *v c* ≠ *v c'* **by** *auto*
hence ∗:*M' 0* (*v c'*) = *min* (*dbm-add* (*M 0* (*v c*)) (*M* (*v c*) (*v c'*))) (*M 0* (*v c'*))
using *A*(*2*,*3*,*6*,*7*) *2* **unfolding** *DBM-reset-def* **by** *auto*
from *2 A*(*2*,*4*) **have** *dbm-entry-val u None* (*Some c'*) (*M' 0* (*v c'*))
unfolding *DBM-val-bounded-def* **by** *auto*
with *dbm-entry-dbm-min2* ∗ **have** *dbm-entry-val u None* (*Some c'*) (*M 0* (*v c'*)) **by** *auto*
thus *?thesis* **using** *False* **by** *cases auto*
next
case *True*
note [*simp*] = *True*[*symmetric*]
show *?thesis*
proof (*cases M 0* (*v c*))

case (*Le t*)
hence $-t \in$ *S-Min-Le* **unfolding** *S-Min-Le* **by** *force*
hence $d' \geq -t$ **using** d' **by** *auto*
thus *?thesis* **using** *A Le* **by** (*auto simp*: *minus-le-iff*)
next
case (*Lt t*)
hence $-t \in$ *S-Min-Lt* **unfolding** *S-Min-Lt* **by** *force*
hence $d' > -t$ **using** d' **by** *auto*
thus *?thesis* **using** *2 Lt* **by** (*auto simp*: *minus-less-iff*)
next
case *INF* **thus** *?thesis* **by** *auto*
qed
qed
next
case (*3 c'*)
show *?case*
proof (*cases* $c = c'$)
case *False*
with *A*(*2*,*7*) **have** $v\ c \neq v\ c'$ **by** *auto*
hence $*$:M' (*v c'*) *0* = *min* (*dbm-add* (*M* (*v c'*) (*v c*)) (*M* (*v c*) *0*)) (*M* (*v c'*) *0*)
using *A*(*2*,*3*,*6*,*7*) *3* **unfolding** *DBM-reset-def* **by** *auto*
from *3 A*(*2*,*4*) **have** *dbm-entry-val u* (*Some c'*) *None* (M' (*v c'*) *0*)
unfolding *DBM-val-bounded-def* **by** *auto*
with *dbm-entry-dbm-min3* $*$ **have** *dbm-entry-val u* (*Some c'*) *None* (*M* (*v c'*) *0*) **by** *auto*
thus *?thesis* **using** *False* **by** *cases auto*
next
case [*symmetric*, *simp*]: *True*
show *?thesis*
proof (*cases M* (*v c*) *0*, *goal-cases*)
case (*1 t*)
hence $t \in$ *S-Max-Le* **unfolding** *S-Max-Le* **by** *force*
hence $d' \leq t$ **using** d' **by** *auto*
thus *?case* **using** *1* **by** (*auto simp*: *minus-le-iff*)
next
case (*2 t*)
hence $t \in$ *S-Max-Lt* **unfolding** *S-Max-Lt* **by** *force*
hence $d' < t$ **using** d' **by** *auto*
thus *?case* **using** *2* **by** (*auto simp*: *minus-less-iff*)
next
case *3* **thus** *?case* **by** *auto*
qed
qed

next
case (*4 c1 c2*)
show *?case*
proof (*cases c = c1*)
case *False*
note *F1 = this*
show *?thesis*
proof (*cases c = c2*)
case *False*
with *A*(*2*,*6*,*7*) *F1* **have** $v\ c \neq v\ c1\ v\ c \neq v\ c2$ **by** *auto*
hence $*$:M' (*v c1*) (*v c2*) = *min* (*dbm-add* (*M* (*v c1*) (*v c*)) (*M* (*v c*) (*v c2*))) (*M* (*v c1*) (*v c2*))
using *A*(*2*,*3*,*6*,*7*) *4* **unfolding** *DBM-reset-def* **by** *auto*
from *4 A*(*2*,*4*) **have** *dbm-entry-val u* (*Some c1*) (*Some c2*) (M' (*v c1*) (*v c2*))
unfolding *DBM-val-bounded-def* **by** *auto*
with *dbm-entry-dbm-min* $*$ **have** *dbm-entry-val u* (*Some c1*) (*Some c2*) (*M* (*v c1*) (*v c2*)) **by** *auto*
thus *?thesis* **using** *F1 False* **by** *cases auto*
next
case [*symmetric*, *simp*]: *True*
show *?thesis*
proof (*cases M* (*v c1*) (*v c*), *goal-cases*)
case (*1 t*)
hence $u\ c1 - t \in$ *S-Min-Le* **unfolding** *S-Min-Le* **using** *A F1 4* **by** *blast*
hence $d' \geq u\ c1 - t$ **using** d' **by** *auto*
hence $t + d' \geq u\ c1$ **by** (*metis le-swap add-le-cancel-right diff-add-cancel*)
hence $u\ c1 - d' \leq t$ **by** (*metis add-le-imp-le-right diff-add-cancel*)
thus *?case* **using** *1 F1* **by** *auto*
next
case (*2 t*)
hence $u\ c1 - t \in$ *S-Min-Lt* **unfolding** *S-Min-Lt* **using** *A 4 F1* **by** *blast*
hence $d' > u\ c1 - t$ **using** d' **by** *auto*
hence $d' + t > u\ c1$ **by** (*metis add-strict-right-mono diff-add-cancel*)
hence $u\ c1 - d' < t$ **by** (*metis gt-swap add-less-cancel-right diff-add-cancel*)
thus *?case* **using** *2 F1* **by** *auto*
next
case *3* **thus** *?case* **by** *auto*
qed
qed

next
case *True*
note $T = this$
show *?thesis*
proof (*cases* $c = c2$)
case *False*
show *?thesis*
proof (*cases* $M\ (v\ c)\ (v\ c2)$, *goal-cases*)
case ($1\ t$)
hence $u\ c2 + t \in S\text{-}Max\text{-}Le$ **unfolding** *S-Max-Le* **using** *A 4 False* **by** *blast*
hence $d' \leq u\ c2 + t$ **using** d' **by** *auto*
hence $d' - u\ c2 \leq t$
by (*metis* (*no-types*) *add-diff-cancel-left add-ac*(*1*) *add-le-cancel-right add-right-cancel diff-add-cancel*)
thus *?case* **using** *1 T False* **by** *auto*
next
case ($2\ t$)
hence $u\ c2 + t \in S\text{-}Max\text{-}Lt$ **unfolding** *S-Max-Lt* **using** *A 4 False* **by** *blast*
hence $d' < u\ c2 + t$ **using** d' **by** *auto*
hence $d' - u\ c2 < t$ **by** (*metis gt-swap add-less-cancel-right diff-add-cancel*)
thus *?case* **using** *2 T False* **by** *force*
next
case *3* **thus** *?case* **using** *T* **by** *auto*
qed
next
case [*symmetric*, *simp*]: *True*
from *A 4* **have** $*$:*dbm-entry-val* u'' (*Some c1*) (*Some c1*) ($M\ (v\ c1)\ (v\ c1)$)
unfolding *DBM-val-bounded-def* **by** *auto*
show *?thesis* **using** *True T*
proof (*cases* $M\ (v\ c1)\ (v\ c1)$, *goal-cases*)
case ($1\ t$)
with $*$ **have** $0 \leq t$ **by** *auto*
thus *?case* **using** *1* **by** *auto*
next
case ($2\ t$)
with $*$ **have** $0 < t$ **by** *auto*
thus *?case* **using** *2* **by** *auto*
next
case *3* **thus** *?case* **by** *auto*
qed

      qed
    qed
  qed
  thus *?thesis* **using** $A(1)$ **by** *blast*
qed

lemma *DBM-reset-sound2*:
  assumes $v\ c \leq n$ *DBM-reset* $M\ n\ (v\ c)\ d\ M'$ *DBM-val-bounded* $v\ u\ M'\ n$
  shows $u\ c = d$
using *assms* **unfolding** *DBM-val-bounded-def DBM-reset-def*
by *fastforce*

lemma *DBM-reset-sound''*:
  fixes $M\ v\ c\ n\ d$
  defines $M' \equiv$ *reset* $M\ n\ (v\ c)\ d$
  assumes *clock-numbering'* $v\ n\ v\ c \leq n$ *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 −
  assume $A{:}\bigwedge d'$. *DBM-val-bounded* $v\ (u(c := d'))\ M\ n \Longrightarrow$ *thesis*
  from *assms DBM-reset-reset*[*of* $v\ c\ n\ M\ d$]
  have $*$:*DBM-reset* $M\ n\ (v\ c)\ d\ M'$ **by** (*auto simp add*: M'-*def*)
  with *DBM-reset-sound'*[*of* $v\ n\ c\ M\ d\ M'$, *OF - - this*] *assms* **obtain** d'
where
  DBM-val-bounded $v\ (u(c := d'))\ M\ n$ **by** *auto*
  with A **show** *thesis* **by** *auto*
qed

lemma *DBM-reset-sound*:
  fixes $M\ v\ c\ n\ d$
  defines $M' \equiv$ *reset* $M\ n\ (v\ c)\ d$
  assumes $\forall\, k \leq n.\ k > 0 \longrightarrow (\exists\, c.\ v\ c = k)$ *clock-numbering'* $v\ n\ v\ c \leq n$
        $u \in [M']_{v,n}$
  obtains d' **where** $u(c := d') \in [M]_{v,n}$
proof (*cases* $[M]_{v,n} = \{\}$)
  case *False*
  then obtain u' **where** *DBM-val-bounded* $v\ u'\ M\ n$ **unfolding** *DBM-zone-repr-def*
by *auto*
  from *DBM-reset-sound''*[*OF assms*$(3{-}4)$ *- this*] *assms*$(1,5)$ *that* **show**
?thesis
  unfolding *DBM-zone-repr-def* **by** *auto*
next
  case *True*
  with *DBM-reset-complete-empty'*[*OF assms*(2) *- - DBM-reset-reset*, *of* v

c M u d] *assms* **show** *?thesis*
  unfolding *DBM-zone-repr-def* **by** *simp*
qed

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*
using *assms*
proof (*induction cs*)
  case *Nil* **thus** *?case* **by** *auto*
next
  case (*Cons c cs*)
  let *?M′* = *reset′ M n cs v d*
  let *?M″* = *reset ?M′ n* (*v c*) *d*
  from *Cons* **obtain** *u′* **where** *u′*: *DBM-val-bounded v u′ ?M′ n* **by** *fastforce*
  from *Cons*(*3*,*4*) **have** *0* < *v c v c* ≤ *n* **by** *auto*
  from *DBM-reset-reset*[*OF this*] **have** ∗∗: *DBM-reset ?M′ n* (*v c*) *d ?M″* **by** *fast*
  from *Cons*(*4*) **have** *v c* ≤ *n* **by** *auto*
  from *DBM-reset-complete*[*of v n c ?M′ d ?M″*, *OF Cons*(*3*) *this* ∗∗ *u′*]
  have *DBM-val-bounded v* (*u′*(*c* := *d*)) (*reset* (*reset′ M n cs v d*) *n* (*v c*) *d*) *n* **by** *fast*
  thus *?case* **by** *auto*
qed

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*
using *assms*
proof (*induction cs*)
  case *Nil* **thus** *?case* **by** *auto*
next
  case (*Cons c cs*)
  let *?M′* = *reset′ M n cs v d*
  let *?M″* = *reset ?M′ n* (*v c*) *d*
  from *Cons* **have** ∗: *DBM-val-bounded v* ([*cs*→*d*]*u*) (*reset′ M n cs v d*) *n* **by** *fastforce*
  from *Cons*(*3*,*4*) **have** *0* < *v c v c* ≤ *n* **by** *auto*
  from *DBM-reset-reset*[*OF this*] **have** ∗∗: *DBM-reset ?M′ n* (*v c*) *d ?M″* **by** *fast*
  from *Cons*(*4*) **have** *v c* ≤ *n* **by** *auto*
  from *DBM-reset-complete*[*of v n c ?M′ d ?M″*, *OF Cons*(*3*) *this* ∗∗ ∗]

have ∗∗∗:*DBM-val-bounded v* ([*c*#*cs*→*d*]*u*) (*reset* (*reset′ M n cs v d*) *n* (*v c*) *d*) *n* **by** *simp*
have *reset′ M n* (*c*#*cs*) *v d* = *reset* (*reset′ M n cs v d*) *n* (*v c*) *d* **by** *auto*
with ∗∗∗ **show** *?case* **by** *presburger*
qed

lemma *DBM-reset′-sound-empty*:
assumes *clock-numbering′ v n* ∀ *c* ∈ *set cs*. *v c* ≤ *n*
∀ *u* . ¬ *DBM-val-bounded v u* (*reset′ M n cs v d*) *n*
shows ¬ *DBM-val-bounded v u M n*
using *assms DBM-reset′-complete* **by** *metis*

fun *set-clocks* :: *′c list* ⇒ *′t*::*time list*⇒ (*′c*,*′t*) *cval* ⇒ (*′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* ∀ *c* ∈ *set cs*. *v c* ≤ *n*
DBM-val-bounded v u (*reset′ M n cs v d*) *n DBM-val-bounded v u′′ M n*
shows ∃ *ts*. *DBM-val-bounded v* (*set-clocks cs ts u*) *M n*
using *assms*
proof (*induction cs arbitrary*: *M u*)
case *Nil*
hence *DBM-val-bounded v* (*set-clocks* [] [] *u*) *M n* **by** *auto*
thus *?case* **by** *blast*
next
case (*Cons c′ cs*)
let *?M′* = *reset′ M n* (*c′* # *cs*) *v d*
let *?M′′* = *reset′ M n cs v d*
from *DBM-reset′-complete*[*OF Cons*(*5*) *Cons*(*2*)] *Cons*(*3*)
have *u′′*: *DBM-val-bounded v* ([*cs*→*d*]*u′′*) *?M′′ n* **by** *fastforce*
from *Cons*(*3*,*4*) **have** *v c′* ≤ *n DBM-val-bounded v u* (*reset ?M′′ n* (*v c′*) *d*) *n* **by** *auto*
from *DBM-reset-sound′′*[*OF Cons*(*2*) *this u′′*]
obtain *d′* **where** ∗∗:*DBM-val-bounded v* (*u*(*c′* := *d′*)) *?M′′ n* **by** *blast*
from *Cons.IH*[*OF Cons.prems*(*1*) - ∗∗ *Cons.prems*(*4*)] *Cons.prems*(*2*)
obtain *ts* **where** *ts*:*DBM-val-bounded v* (*set-clocks cs ts* (*u*(*c′* := *d′*))) *M n* **by** *fastforce*
hence *DBM-val-bounded v* (*set-clocks* (*c′* # *cs*) (*d′*#*ts*) *u*) *M n* **by** *auto*
thus *?case* **by** *fast*

qed

lemma *DBM-reset'-resets*:
fixes *M v c n d cs*
assumes $\forall$ *k*$\le$*n*. *k* $>$ *0* $\longrightarrow$ ($\exists$ *c*. *v c* = *k*) *clock-numbering' v n* $\forall$ *c* $\in$ *set cs*. *v c* $\le$ *n*
DBM-val-bounded v u (*reset' M n cs v d*) *n*
shows $\forall$ *c* $\in$ *set cs*. *u c* = *d*
using *assms*
proof (*induction cs arbitrary*: *M u*)
case *Nil* **thus** *?case* **by** *auto*
next
case (*Cons c' cs*)
let *?M'* = *reset' M n* (*c'* # *cs*) *v d*
let *?M''* = *reset' M n cs v d*
from *Cons*(*4*,*5*) **have** *v c'* $\le$ *n DBM-val-bounded v u* (*reset ?M'' n* (*v c'*) *d*) *n* **by** *auto*
from *DBM-reset-sound2*[*OF this*(*1*) - *Cons*(*5*), *of ?M'' d*] *DBM-reset-reset*[*OF* - *this*(*1*), *of ?M'' d*] *Cons*(*3*)
have *c'*:*u c'* = *d* **by** *auto*
from *Cons*(*4*,*5*) **have** *v c'* $\le$ *n DBM-val-bounded v u* (*reset ?M'' n* (*v c'*) *d*) *n* **by** *auto*
with *DBM-reset-sound*[*OF Cons.prems*(*1*,*2*) *this*(*1*)]
obtain *d'* **where** ∗∗:*DBM-val-bounded v* (*u*(*c'* := *d'*)) *?M'' n* **unfolding** *DBM-zone-repr-def* **by** *blast*
from *Cons.IH*[*OF Cons.prems*(*1*,*2*) - ∗∗] *Cons.prems*(*3*) **have** $\forall$ *c*$\in$*set cs*. (*u*(*c'* := *d'*)) *c* = *d* **by** *auto*
thus *?case* **using** *c'*
by (*auto split*: *if-split-asm*)
qed

lemma *DBM-reset'-resets'*:
fixes *M* :: (*'t* :: *time*) *DBM* **and** *v c n d cs*
assumes *clock-numbering' v n* $\forall$ *c* $\in$ *set cs*. *v c* $\le$ *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*
using *assms*
proof (*induction cs arbitrary*: *M u*)
case *Nil* **thus** *?case* **by** *auto*
next
case (*Cons c' cs*)
let *?M'* = *reset' M n* (*c'* # *cs*) *v d*
let *?M''* = *reset' M n cs v d*

from *DBM-reset′-complete*[*OF Cons*(*5*) *Cons*(*2*)] *Cons*(*3*)
have *u″*: *DBM-val-bounded v* ([*cs*→*d*]*u″*) *?M″ n* **by** *fastforce*
from *Cons*(*3*,*4*) **have** *v c′* ≤ *n DBM-val-bounded v u* (*reset ?M″ n* (*v c′*) *d*) *n* **by** *auto*
from *DBM-reset-sound2*[*OF this*(*1*) - *Cons*(*4*), *of ?M″ d*] *DBM-reset-reset*[*OF* - *this*(*1*), *of ?M″ d*] *Cons*(*2*)
have *c′*:*u c′* = *d* **by** *auto*
from *Cons*(*3*,*4*) **have** *v c′* ≤ *n DBM-val-bounded v u* (*reset ?M″ n* (*v c′*) *d*) *n* **by** *auto*
from *DBM-reset-sound″*[*OF Cons*(*2*) *this u″*]
obtain *d′* **where** ∗∗:*DBM-val-bounded v* (*u*(*c′* := *d′*)) *?M″ n* **by** *blast*
from *Cons.IH*[*OF Cons.prems*(*1*) - ∗∗ *Cons.prems*(*4*)] *Cons.prems*(*2*)
have ∀ *c*∈*set cs*. (*u*(*c′* := *d′*)) *c* = *d* **by** *auto*
thus *?case* **using** *c′*
by (*auto split*: *if-split-asm*)
qed

lemma *DBM-reset′-neg-diag-preservation′*:
fixes *M* :: (*′t* :: *time*) *DBM*
assumes *k*≤*n M k k* < *0 clock-numbering v* ∀ *c* ∈ *set cs*. *v c* ≤ *n*
shows *reset′ M n cs v d k k* < *0* **using** *assms*
proof (*induction cs*)
case *Nil* **thus** *?case* **by** *auto*
next
case (*Cons c cs*)
then have *IH*: *reset′ M n cs v d k k* < *0* **by** *auto*
from *Cons.prems* **have** *v c* > *0 v c* ≤ *n* **by** *auto*
from *DBM-reset-reset*[*OF this*, *of reset′ M n cs v d d*] ‹*k* ≤ *n*›
have *reset* (*reset′ M n cs v d*) *n* (*v c*) *d k k* ≤ *reset′ M n cs v d k k*
unfolding *DBM-reset-def*
by (*cases v c* = *k*, *cases k* = *0*, *auto simp*: *less*[*symmetric*])
with *IH* **show** *?case* **by** *auto*
qed

lemma *DBM-reset′-complete-empty′*:
assumes ∀ *k*≤*n*. *k* > *0* ⟶ (∃ *c*. *v c* = *k*) *clock-numbering′ v n*
∀ *c* ∈ *set cs*. *v c* ≤ *n* ∀ *u* . ¬ *DBM-val-bounded v u M n*
shows ∀ *u* . ¬ *DBM-val-bounded v u* (*reset′ M n cs v d*) *n* **using** *assms*
proof (*induction cs*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons c cs*)
then have ∀ *u*. ¬ *DBM-val-bounded v u* (*reset′ M n cs v d*) *n* **by** *auto*
from *Cons.prems*(*2*,*3*) *DBM-reset-complete-empty′*[*OF Cons.prems*(*1*) -

- *DBM-reset-reset this*]
  show *?case* **by** *auto*
qed

lemma *DBM-reset′-complete-empty*:
  assumes ∀ *k*≤*n*. *k* > *0* ⟶ (∃ *c*. *v c* = *k*) *clock-numbering′ v n*
    ∀ *c* ∈ *set cs*. *v c* ≤ *n* ∀ *u* . ¬ *DBM-val-bounded v u M n*
  shows ∀ *u* . ¬ *DBM-val-bounded v u* (*reset′* (*FW M n*) *n cs v d*) *n* **using** *assms*
proof −
  note *A* = *assms*
  from *A*(*4*) **have** $[M]_{v,n}$ = {} **unfolding** *DBM-zone-repr-def* **by** *auto*
  with *FW-zone-equiv*[*OF A*(*1*)] **have** $[FW\ M\ n]_{v,n}$ = {} **by** *auto*
  with *FW-detects-empty-zone*[*OF A*(*1*)] *A*(*2*) **obtain** *i* **where** *i*: *i* ≤ *n* *FW M n i i* < *Le 0* **by** *blast*
  with *DBM-reset′-neg-diag-preservation′ A*(*2*,*3*) **have**
    reset′ (*FW M n*) *n cs v d i i* < *Le 0*
  by (*auto simp*: *neutral*)
  with *fw-mono*[*of i n i reset′* (*FW M n*) *n cs v d n*] *i*
  have *FW* (*reset′* (*FW M n*) *n cs v d*) *n i i* < *Le 0* **by** *auto*
  with *FW-detects-empty-zone*[*OF A*(*1*), *of reset′* (*FW M n*) *n cs v d*] *A*(*2*,*3*) *i*
  have $[FW\ (reset'\ (FW\ M\ n)\ n\ cs\ v\ d)\ n]_{v,n}$ = {} **by** *auto*
  with *FW-zone-equiv*[*OF A*(*1*), *of reset′* (*FW M n*) *n cs v d*] *A*(*3*,*4*)
  show *?thesis* **by** (*auto simp*: *DBM-zone-repr-def*)
qed

lemma *DBM-reset′-empty′*:
  assumes ∀ *k*≤*n*. *k* > *0* ⟶ (∃ *c*. *v c* = *k*) *clock-numbering′ v n* ∀ *c* ∈ *set cs*. *v c* ≤ *n*
  shows $[M]_{v,n}$ = {} ⟷ $[reset'\ (FW\ M\ n)\ n\ cs\ v\ d]_{v,n}$ = {}
proof
  let *?M′* = *reset′* (*FW M n*) *n cs v d*
  assume *A*: $[M]_{v,n}$ = {}
  hence ∀ *u* . ¬ *DBM-val-bounded v u M n* **unfolding** *DBM-zone-repr-def* **by** *auto*
  with *DBM-reset′-complete-empty*[*OF assms*] **show** $[?M']_{v,n}$ = {} **unfolding** *DBM-zone-repr-def* **by** *auto*
next
  let *?M′* = *reset′* (*FW M n*) *n cs v d*
  assume *A*: $[?M']_{v,n}$ = {}
  hence ∀ *u* . ¬ *DBM-val-bounded v u ?M′ n* **unfolding** *DBM-zone-repr-def* **by** *auto*
  from *DBM-reset′-sound-empty*[*OF assms*(*2*,*3*) *this*] **have** ∀ *u*. ¬ *DBM-val-bounded*

v u (*FW M n*) *n* **by** *auto*
with *FW-zone-equiv*[*OF assms*(*1*)] **show** $[M]_{v,n} = \{\}$ **unfolding** *DBM-zone-repr-def* **by** *auto*
qed

lemma *DBM-reset′-empty*:
assumes $\forall k \le n.\ k > 0 \longrightarrow (\exists c.\ v\ c = k)$ *clock-numbering′ v n* $\forall\ c \in set\ cs.\ v\ c \le n$
shows $[M]_{v,n} = \{\} \longleftrightarrow [reset'\ M\ n\ cs\ v\ d]_{v,n} = \{\}$
proof
let *?M′* = *reset′ M n cs v d*
assume *A*: $[M]_{v,n} = \{\}$
hence $\forall\ u\ .\ \neg$ *DBM-val-bounded v u M n* **unfolding** *DBM-zone-repr-def* **by** *auto*
with *DBM-reset′-complete-empty′*[*OF assms*] **show** $[?M']_{v,n} = \{\}$ **unfolding** *DBM-zone-repr-def* **by** *auto*
next
let *?M′* = *reset′ M n cs v d*
assume *A*: $[?M']_{v,n} = \{\}$
hence $\forall\ u\ .\ \neg$ *DBM-val-bounded v u ?M′ n* **unfolding** *DBM-zone-repr-def* **by** *auto*
from *DBM-reset′-sound-empty*[*OF assms*(*2*,*3*) *this*] **have** $\forall\ u.\ \neg$ *DBM-val-bounded v u M n* **by** *auto*
with *FW-zone-equiv*[*OF assms*(*1*)] **show** $[M]_{v,n} = \{\}$ **unfolding** *DBM-zone-repr-def* **by** *auto*
qed

lemma *DBM-reset′-sound*:
assumes $\forall k \le n.\ k > 0 \longrightarrow (\exists c.\ v\ c = k)$ *clock-numbering′ v n*
and $\forall c \in set\ cs.\ v\ c \le n$
and $u \in [reset'\ M\ n\ cs\ v\ d]_{v,n}$
shows $\exists ts.$ *set-clocks cs ts u* $\in [M]_{v,n}$
proof −
from *DBM-reset′-empty*[*OF assms*(*1*−*3*)] *assms*(*4*) **obtain** *u′* **where** $u' \in [M]_{v,n}$ **by** *blast*
with *DBM-reset′-sound′*[*OF assms*(*2*,*3*)] *assms*(*4*) **show** *?thesis* **unfolding** *DBM-zone-repr-def* **by** *blast*
qed

3.5 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* $(a + b) \in \mathbb{Z}$

by (*cases a*) (*cases b*, *auto simp*: *add*)+

lemma *sum-not-inf-dest*:
  assumes $a + b \neq (\infty :: - DBMEntry)$
  shows $a \neq (\infty :: - DBMEntry) \wedge b \neq (\infty :: - DBMEntry)$
using *assms* **by** (*cases a*; *cases b*; *simp add*: *add*)

lemma *sum-not-inf-int*:
  assumes $a + b \neq (\infty :: - DBMEntry)$ *get-const* $a \in \mathbb{Z}$ *get-const* $b \in \mathbb{Z}$
  shows *get-const* $(a + b) \in \mathbb{Z}$
using *assms sum-not-inf-dest* **by** *fastforce*

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 (*goal-cases*)
  case *1*
  show *?thesis*
  proof (*cases* $i = i' \wedge j = j'$)
    case *True*
     with *1* **show** *?thesis* **by** (*fastforce simp*: *fw-upd-def upd-def min-def dest*: *sum-not-inf-dest*)
  next
    case *False*
    with *1* **show** *?thesis* **by** (*auto simp* : *fw-upd-def upd-def*)
  qed
qed

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

abbreviation *dbm-int-all* $M \equiv \forall\ i.\ \forall\ j.\ M\ i\ j \neq \infty \longrightarrow$ *get-const* $(M\ i\ j) \in \mathbb{Z}$

lemma *dbm-intI*:
  dbm-int-all $M \Longrightarrow$ *dbm-int* $M\ n$
by *auto*

lemma *fwi-int-preservation*:
  dbm-int (*fwi* $M\ n\ k\ i\ j$) n **if** *dbm-int* $M\ n\ k \leq n$
  apply (*induction* - (i, j) *arbitrary*: $i\ j$ *rule*: *wf-induct*[*of less-than* <∗*lex*∗> *less-than*])

apply *force*
subgoal for *i j*
using *that*
by (*cases i*; *cases j*) (*auto 4 3 dest*: *sum-not-inf-dest simp*: *min-def fw-upd-def upd-def*)
done

lemma *fw-int-preservation*:
dbm-int (*fw M n k*) *n* **if** *dbm-int M n k* $\leq$ *n*
using ‹*k* $\leq$ *n*› **apply** (*induction k*)
using *that* **apply** *simp*
apply (*rule fwi-int-preservation*; *auto*)
using *that* **by** (*simp*) (*rule fwi-int-preservation*; *auto*)

lemma *FW-int-preservation*:
assumes *dbm-int M n*
shows *dbm-int* (*FW M n*) *n*
using *fw-int-preservation*[*OF assms*(*1*)] **by** *auto*

lemma *FW-int-all-preservation*:
assumes *dbm-int-all M*
shows *dbm-int-all* (*FW M n*)
using *assms*
apply *clarify*
subgoal for *i j*
apply (*cases i* $\leq$ *n*)
apply (*cases j* $\leq$ *n*)
by (*auto simp*: *FW-int-preservation*[*OF dbm-intI*[*OF assms*(*1*)]] *FW-out-of-bounds1 FW-out-of-bounds2*)
done

lemma *And-int-all-preservation*[*intro*]:
assumes *dbm-int-all M1 dbm-int-all M2*
shows *dbm-int-all* (*And M1 M2*)
using *assms* **by** (*auto simp*: *min-def*)

lemma *And-int-preservation*:
assumes *dbm-int M1 n dbm-int M2 n*
shows *dbm-int* (*And M1 M2*) *n*
using *assms* **by** (*auto simp*: *min-def*)

lemma *up-int-all-preservation*:
dbm-int-all (*M* :: ((*'t* :: {*time*, *ring-1*}) *DBM*)) $\Longrightarrow$ *dbm-int-all* (*up M*)
unfolding *up-def min-def add*[*symmetric*] **by** (*auto dest*: *sum-not-inf-dest*

split: *if-split-asm*)

lemma *up-int-preservation*:
  dbm-int (M :: (('t :: {*time*, *ring-1*}) *DBM*)) $n \Longrightarrow$ *dbm-int* (*up* M) n
  unfolding *up-def min-def add*[*symmetric*] **by** (*auto dest*: *sum-not-inf-dest* *split*: *if-split-asm*)

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 *clarify*
  fix i j
  assume A: $i \leq n$ $j \leq n$ M' i $j \neq \infty$
  from *assms*(*2*) **show** *get-const* (M' i j) $\in \mathbb{Z}$ **unfolding** *DBM-reset-def*
    apply (*cases* $i = k$; *cases* $j = k$)
       apply *simp*
    subgoal using A *assms*(*1*,*4*) **by** *presburger*
      apply (*cases* $j = 0$)
    subgoal using *assms*(*3*) **by** *simp*
    subgoal using A **by** *simp*
     apply *simp*
     apply (*cases* $i = 0$)
    subgoal using *assms*(*3*) **by** *simp*
    subgoal using A **by** *simp*
    using A **apply** *simp*
    apply (*simp split*: *split-min*, *safe*)
    subgoal
    proof *goal-cases*
      case *1*
      then have $*$: M i k + M k $j \neq \infty$ **unfolding** *add min-def* **by** *meson*
      with *sum-not-inf-dest* **have** M i $k \neq \infty$ M k $j \neq \infty$ **by** *auto*
       with *1*(*3*,*4*) *assms*(*1*,*4*) **have** *get-const* (M i k) $\in \mathbb{Z}$ *get-const* (M k j) $\in \mathbb{Z}$ **by** *auto*
        with *sum-not-inf-int*[*folded add*, *OF* $*$] **show** *?case* **unfolding** *add* **by** *auto*
    qed
    subgoal
    proof *goal-cases*
      case *1*
      then have $*$: M i $j \neq \infty$ **unfolding** *add min-def* **by** *meson*
      with *1*(*3*,*4*) *assms*(*1*,*4*) **show** *?case* **by** *auto*
    qed
  done

qed

lemma *DBM-reset-int-preservation*:
  fixes *M* :: (*'t* :: {*time*,*ring-1*}) *DBM*
  assumes *dbm-int M n d* ∈ ℤ *0* < *k k* ≤ *n*
  shows *dbm-int* (*reset M n k d*) *n*
using *assms*(*3*−) *DBM-reset-int-preservation'*[*OF assms*(*1*) *DBM-reset-reset assms*(*2*)] **by** *blast*

lemma *DBM-reset-int-all-preservation*:
  fixes *M* :: (*'t* :: {*time*,*ring-1*}) *DBM*
  assumes *dbm-int-all M d* ∈ ℤ
  shows *dbm-int-all* (*reset M n k d*)
using *assms*
 apply *clarify*
 subgoal for *i j*
   by (*cases i* = *k*; *cases j* = *k*;
      auto simp: *reset-def min-def add*[*symmetric*] *dest*!: *sum-not-inf-dest*
)
done

lemma *DBM-reset'-int-all-preservation*:
  fixes *M* :: (*'t* :: {*time*, *ring-1*}) *DBM*
  assumes *dbm-int-all M d* ∈ ℤ
  shows *dbm-int-all* (*reset' M n cs v d*) **using** *assms*
by (*induction cs*) (*simp* | *rule DBM-reset-int-all-preservation*)+

lemma *DBM-reset'-int-preservation*:
  fixes *M* :: (*'t* :: {*time*, *ring-1*}) *DBM*
  assumes *dbm-int M n d* ∈ ℤ ∀ *c*. *v c* > *0* ∀ *c* ∈ *set cs*. *v c* ≤ *n*
  shows *dbm-int* (*reset' M n cs v d*) *n* **using** *assms*
proof (*induction cs*)
  case *Nil* **then show** *?case* **by** *simp*
next
  case (*Cons c cs*)
 from *Cons.IH*[*OF Cons.prems*(*1*,*2*,*3*)] *Cons.prems*(*4*) **have** *dbm-int* (*reset' M n cs v d*) *n*
    by *fastforce*
 from *DBM-reset-int-preservation*[*OF this Cons.prems*(*2*), *of v c*] *Cons.prems*(*3*,*4*)
show *?case*
    by *auto*
qed

lemma *reset-set1*:

$\forall\, c \in set\ cs.\ ([cs \to d]u)\ c = d$
by (*induction cs*) *auto*

lemma *reset-set11*:
$\forall\, c.\ c \notin set\ cs \longrightarrow ([cs \to d]u)\ c = u\ c$
by (*induction cs*) *auto*

lemma *reset-set2*:
$\forall\, c.\ c \notin set\ cs \longrightarrow (set\text{-}clocks\ cs\ ts\ u)c = u\ c$
proof (*induction cs arbitrary*: *ts u*)
case *Nil* **then show** *?case* **by** *auto*
next
case *Cons* **then show** *?case*
proof (*cases ts*, *goal-cases*)
case *Nil* **then show** *?thesis* **by** *simp*
next
case (*2 a'*) **then show** *?case* **by** *auto*
qed
qed

lemma *reset-set*:
assumes $\forall\ c \in set\ cs.\ u\ c = d$
shows $[cs \to d](set\text{-}clocks\ cs\ ts\ u) = u$
proof
fix *c*
show $([cs \to d]set\text{-}clocks\ cs\ ts\ u)\ c = u\ c$
proof (*cases* $c \in set\ cs$)
case *True*
hence $([cs \to d]set\text{-}clocks\ cs\ ts\ u)\ c = d$ **using** *reset-set1* **by** *fast*
also have $d = u\ c$ **using** *assms True* **by** *auto*
finally show *?thesis* **by** *auto*
next
case *False*
hence $([cs \to d]set\text{-}clocks\ cs\ ts\ u)\ c = set\text{-}clocks\ cs\ ts\ u\ c$ **by** (*simp add*: *reset-set11*)
also with *False* **have** $\ldots = u\ c$ **by** (*simp add*: *reset-set2*)
finally show *?thesis* **by** *auto*
qed
qed

3.5.1 Unused theorems

lemma *canonical-cyc-free*:
$canonical\ M\ n \Longrightarrow \forall\, i \leq n.\ M\ i\ i \geq 0 \Longrightarrow cyc\text{-}free\ M\ n$

by (*auto dest*!: *canonical-len*)

lemma *canonical-cyc-free2*:
canonical M n $\Longrightarrow$ *cyc-free M n* $\longleftrightarrow$ ($\forall$ *i* $\leq$ *n*. *M i i* $\geq$ *0*)
apply *safe*
apply (*simp add*: *cyc-free-diag-dest'*)
using *canonical-cyc-free* **by** *blast*

lemma *DBM-reset'-diag-preservation*:
fixes *M* :: (*'t* :: *time*) *DBM*
assumes $\forall$ *k*$\leq$*n*. *M k k* $\leq$ *0 clock-numbering v* $\forall$ *c* $\in$ *set cs*. *v c* $\leq$ *n*
shows $\forall$ *k*$\leq$*n*. *reset' M n cs v d k k* $\leq$ *0* **using** *assms*
proof (*induction cs*)
case *Nil* **thus** *?case* **by** *auto*
next
case (*Cons c cs*)
then have *IH*: $\forall$ *k*$\leq$*n*. *reset' M n cs v d k k* $\leq$ *0* **by** *auto*
from *Cons.prems* **have** *v c* > *0 v c* $\leq$ *n* **by** *auto*
from *DBM-reset-diag-preservation*[*of n reset' M n cs v d*, *OF IH DBM-reset-reset*, *of v c*, *OF this*]
show *?case* **by** *simp*
qed

end
theory *DBM-Misc*
imports
Main
HOL.Real
begin

lemma *finite-set-of-finite-funs2*:
fixes *A* :: *'a set*
and *B* :: *'b set*
and *C* :: *'c set*
and *d* :: *'c*
assumes *finite A*
and *finite B*
and *finite C*
shows *finite* {*f*. $\forall$ *x*. $\forall$ *y*. (*x* $\in$ *A* $\wedge$ *y* $\in$ *B* $\longrightarrow$ *f x y* $\in$ *C*) $\wedge$ (*x* $\notin$ *A* $\longrightarrow$ *f x y* = *d*) $\wedge$ (*y* $\notin$ *B* $\longrightarrow$ *f x y* = *d*)}
proof –
let *?S* = {*f*. $\forall$ *x*. $\forall$ *y*. (*x* $\in$ *A* $\wedge$ *y* $\in$ *B* $\longrightarrow$ *f x y* $\in$ *C*) $\wedge$ (*x* $\notin$ *A* $\longrightarrow$ *f x y* = *d*) $\wedge$ (*y* $\notin$ *B* $\longrightarrow$ *f x y* = *d*)}
let *?R* = {*g*. $\forall$ *x*. (*x* $\in$ *B* $\longrightarrow$ *g x* $\in$ *C*) $\wedge$ (*x* $\notin$ *B* $\longrightarrow$ *g x* = *d*)}

let *?Q* = {*f*. ∀ *x*. (*x* ∈ *A* ⟶ *f x* ∈ *?R*) ∧ (*x* ∉ *A* ⟶ *f x* = (λ*y*. *d*))}
from *finite-set-of-finite-funs*[*OF assms*(*2*,*3*)] **have** *finite ?R* **.**
from *finite-set-of-finite-funs*[*OF assms*(*1*) *this*, *of* λ *y*. *d*] **have** *finite ?Q*
.
moreover have *?S* = *?Q*
by *force+*
ultimately show *?thesis* **by** *simp*
qed

end

3.6 Extrapolation of DBMs

theory *DBM-Normalization*
imports
DBM-Basics
DBM-Misc
HOL−Eisbach.Eisbach
begin

NB: The journal paper on extrapolations based on lower and upper bounds [1] provides slightly incorrect definitions that would always set (lower) bounds of the form *M 0 i* to ∞. To fix this, we use two invariants that can also be found in TChecker's DBM library, for instance:

1. Lower bounds are always nonnegative, i.e. ∀ *i* ≤ *n*. *M 0 i* ≤ *0* (see *extra-lup-lower-bounds*).
2. Entries to the diagonal is always normalized to *Le 0*, *Lt 0* or ∞. This makes it again obvious that the set of normalized DBMs is finite.

lemmas *dbm-less-simps*[*simp*] = *dbm-lt-code-simps*[*folded DBM.less*]

lemma *dbm-less-eq-simps*[*simp*]:
Le a ≤ *Le b* ⟷ *a* ≤ *b*
Le a ≤ *Lt b* ⟷ *a* < *b*
Lt a ≤ *Le b* ⟷ *a* ≤ *b*
Lt a ≤ *Lt b* ⟷ *a* ≤ *b*
unfolding *less-eq dbm-le-def* **by** *auto*

lemma *Le-less-Lt*[*simp*]: *Le x* < *Lt x* ⟷ *False*
using *leD* **by** *blast*

3.6.1 Classical extrapolation

This is the implementation of the classical extrapolation operator ($Extra_M$).

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

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

definition
  norm-diag e = (*if e* ≺ *Le 0 then Lt 0 else if e* = *Le 0 then e else* ∞)

Note that literature pretends that **0** would have a bound of negative infinity 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 :: *linordered-ab-group-add*) *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*
      if i ≠ *j then norm-lower* (*norm-upper* (*M i j*) *ub*) *lb else norm-diag* (*M i j*)
    else M i j

3.6.2 Extrapolations based on lower and upper bounds

This is the implementation of the LU-bounds based extrapolation operation (*Extra*-{*LU*}).

definition *extra-lu* ::
  ('t :: *linordered-ab-group-add*) *DBM* ⇒ (*nat* ⇒ 't) ⇒ (*nat* ⇒ 't) ⇒ *nat* ⇒ 't *DBM*
where
  extra-lu M l u n ≡ λ*i j*.
    let ub = *if i* > *0 then l i*   *else 0 in*
    let lb = *if j* > *0 then* − *u j else 0 in*
    if i ≤ *n* ∧ *j* ≤ *n then*
      if i ≠ *j then norm-lower* (*norm-upper* (*M i j*) *ub*) *lb else norm-diag* (*M i j*)
    else M i j

lemma *norm-is-extra*:

norm M k n = *extra-lu M k k n*
unfolding *norm-def extra-lu-def* **..**

This is the implementation of the LU-bounds based extrapolation operation (*Extra*-{*LU*}^+).

definition *extra-lup* ::
(′*t* :: *linordered-ab-group-add*) *DBM* ⇒ (*nat* ⇒ ′*t*) ⇒ (*nat* ⇒ ′*t*) ⇒ *nat* ⇒ ′*t DBM*
where

```
extra-lup M l u n ≡ λi j.
  let ub = if i > 0 then Lt (l i) else Le 0;
      lb = if j > 0 then Lt (− u j) else Lt 0
  in
  if i ≤ n ∧ j ≤ n then
    if i ≠ j then
      if ub ≺ M i j then ∞
      else if i > 0 ∧ M 0 i ≺ Lt (− l i) then ∞
      else if i > 0 ∧ M 0 j ≺ lb then ∞
      else if i = 0 ∧ M 0 j ≺ lb then Lt (− u j)
      else M i j
    else norm-diag (M i j)
  else M i j
```

method *csimp* = (*clarsimp simp*: *extra-lup-def Let-def DBM.less*[*symmetric*] *not-less any-le-inf neutral*)

method *solve* = *csimp?*; *safe?*; (*csimp* | *meson Lt-le-LeI le-less le-less-trans less-asym*′); *fail*

~~lemma extrapolations-diag-preservation: extra-lu M L U n i i = M i i extra-lup M L U n i i = M i i norm M k n i i = M i i unfolding extra-lu-def extra-lup-def norm-def Let-def by auto~~

lemma
assumes ∀ *i* ≤ *n*. *i* > *0* ⟶ *M 0 i* ≤ *0* ∀ *i* ≤ *n*. *U i* ≥ *0*
shows
extra-lu-lower-bounds: ∀ *i* ≤ *n*. *i* > *0* ⟶ *extra-lu M L U n 0 i* ≤ *0*
and
norm-lower-bounds: ∀ *i* ≤ *n*. *i* > *0* ⟶ *norm M U n 0 i* ≤ *0* **and**
extra-lup-lower-bounds: ∀ *i* ≤ *n*. *i* > *0* ⟶ *extra-lup M L U n 0 i* ≤ *0*
using *assms* **unfolding** *extra-lu-def norm-def* **by** − (*csimp*; *force*)+

lemma *extra-lu-le-extra-lup*:

assumes *canonical*: *canonical M n*
and *canonical-lower-bounds*: $\forall\, i \le n.\ i > 0 \longrightarrow M\ 0\ i \le 0$
shows *extra-lu M l u n i j* $\le$ *extra-lup M l u n i j*
proof −
have $M\ 0\ j \le M\ i\ j$ **if** $i \le n\ j \le n\ i > 0$
proof −
have $M\ 0\ i \le 0$
using *canonical-lower-bounds* ‹$i \le n$› ‹$i > 0$› **by** *simp*
then have $M\ 0\ i + M\ i\ j \le M\ i\ j$
by (*simp add*: *add-decreasing*)
also have $M\ 0\ j \le M\ 0\ i + M\ i\ j$
using *canonical that* **by** *auto*
finally (*xtrans*) **show** *?thesis* **.**
qed
then show *?thesis*
unfolding *extra-lu-def Let-def* **by** (*cases* $i \le n$; *cases* $j \le n$) (*simp*; *safe?*; *solve*)+
qed

lemma *extra-lu-subs-extra-lup*:
assumes *canonical*: *canonical M n* **and** *canonical-lower-bounds*: $\forall\, i \le n.\ i > 0 \longrightarrow M\ 0\ i \le 0$
shows $[$*extra-lu M L U n*$]_{v,n} \subseteq [$*extra-lup M L U n*$]_{v,n}$
using *assms*
by (*auto intro*: *extra-lu-le-extra-lup simp*: *DBM.less-eq*[*symmetric*] *elim*!: *DBM-le-subset*[*rotated*])

3.6.3 Extrapolations are widening operators

lemma *extra-lu-mono*:
assumes $\forall\, c.\ v\ c > 0\ u \in [M]_{v,n}$
shows $u \in [$*extra-lu M L U n*$]_{v,n}$ (**is** $u \in [?M2]_{v,n}$)
proof −
note *A* = *assms*
note *M1* = *A*(*2*)[*unfolded DBM-zone-repr-def DBM-val-bounded-def*]
show *?thesis*
unfolding *DBM-zone-repr-def DBM-val-bounded-def*
proof *safe*
show *Le 0* $\preceq$ *?M2 0 0*
using *A* **unfolding** *extra-lu-def DBM-zone-repr-def DBM-val-bounded-def dbm-le-def norm-diag-def*
by *auto*
next
fix *c* **assume** $v\ c \le n$

with *M1* **have** *M1*: *dbm-entry-val u None* (*Some c*) (*M 0* (*v c*)) **by** *auto*
from ‹*v c* ≤ *n*› *A* **have** ∗:
?M2 0 (*v c*) = *norm-lower* (*norm-upper* (*M 0* (*v c*)) *0*) (− *U* (*v c*))
unfolding *extra-lu-def* **by** *auto*
show *dbm-entry-val u None* (*Some c*) (*?M2 0* (*v c*))
proof (*cases M 0* (*v c*) ≺ *Lt* (− *U* (*v c*)))
case *True*
show *?thesis*
proof (*cases Le 0* ≺ *M 0* (*v c*))
case *True* **with** ∗ **show** *?thesis* **by** *auto*
next
case *False*
with ∗ *True* **have** *?M2 0* (*v c*) = *Lt* (− *U* (*v c*)) **by** *auto*
moreover from *True dbm-entry-val-mono2*[*OF M1*] **have**
dbm-entry-val u None (*Some c*) (*Lt* (− *U* (*v c*)))
by *auto*
ultimately show *?thesis* **by** *auto*
qed
next
case *False*
show *?thesis*
proof (*cases Le 0* ≺ *M 0* (*v c*))
case *True* **with** ∗ **show** *?thesis* **by** *auto*
next
case *F*: *False*
with *M1* ∗ *False* **show** *?thesis* **by** *auto*
qed
qed
next
fix *c* **assume** *v c* ≤ *n*
with *M1* **have** *M1*: *dbm-entry-val u* (*Some c*) *None* (*M* (*v c*) *0*) **by** *auto*
from ‹*v c* ≤ *n*› *A* **have** ∗:
?M2 (*v c*) *0* = *norm-lower* (*norm-upper* (*M* (*v c*) *0*) (*L* (*v c*))) *0*
unfolding *extra-lu-def* **by** *auto*
show *dbm-entry-val u* (*Some c*) *None* (*?M2* (*v c*) *0*)
proof (*cases Le* (*L* (*v c*)) ≺ *M* (*v c*) *0*)
case *True*
with *A*(*1*,*2*) ‹*v c* ≤ *n*› **have** *?M2* (*v c*) *0* = ∞ **unfolding** *extra-lu-def* **by** *auto*
then show *?thesis* **by** *auto*
next
case *False*

show *?thesis*
proof (*cases M* (*v c*) *0* ≺ *Lt 0*)
case *True* **with** *False* ∗ *dbm-entry-val-mono3*[*OF M1*] **show** *?thesis* **by** *auto*
next
case *F*: *False*
with *M1* ∗ *False* **show** *?thesis* **by** *auto*
qed
qed
next
fix *c1 c2* **assume** *v c1* ≤ *n v c2* ≤ *n*
with *M1* **have** *M1*: *dbm-entry-val u* (*Some c1*) (*Some c2*) (*M* (*v c1*) (*v c2*)) **by** *auto*
show *dbm-entry-val u* (*Some c1*) (*Some c2*) (*?M2* (*v c1*) (*v c2*))
proof (*cases v c1* = *v c2*)
case *True*
with *M1* **show** *?thesis*
by (*auto simp*: *extra-lu-def norm-diag-def dbm-entry-val.simps dbm-lt.simps*)
(*meson diff-less-0-iff-less le-less-trans less-le-trans*)+
next
case *False*
show *?thesis*
proof (*cases Le* (*L* (*v c1*)) ≺ *M* (*v c1*) (*v c2*))
case *True*
with *A*(*1*,*2*) ‹*v c1* ≤ *n*› ‹*v c2* ≤ *n*› ‹*v c1* ≠ *v c2*› **have** *?M2* (*v c1*) (*v c2*) = ∞
unfolding *extra-lu-def* **by** *auto*
then show *?thesis* **by** *auto*
next
case *False*
with *A*(*1*,*2*) ‹*v c1* ≤ *n*› ‹*v c2* ≤ *n*› ‹*v c1* ≠ *v c2*› **have** ∗:
?M2 (*v c1*) (*v c2*) = *norm-lower* (*M* (*v c1*) (*v c2*)) (− *U* (*v c2*))
unfolding *extra-lu-def* **by** *auto*
show *?thesis*
proof (*cases M* (*v c1*) (*v c2*) ≺ *Lt* (− *U* (*v c2*)))
case *True*
with *dbm-entry-val-mono1*[*OF M1*] **have**
dbm-entry-val u (*Some c1*) (*Some c2*) (*Lt* (− *U* (*v c2*)))
by *auto*
then have *u c1* − *u c2* < − *U* (*v c2*) **by** *auto*
with ∗ *True* **show** *?thesis* **by** *auto*
next
case *False* **with** *M1* ∗ **show** *?thesis* **by** *auto*
qed

qed
qed
qed
qed

lemma *norm-mono*:
assumes $\forall$ *c. v c > 0 u* $\in$ $[M]_{v,n}$
shows *u* $\in$ $[norm\ M\ k\ n]_{v,n}$
using *assms* **unfolding** *norm-is-extra* **by** (*rule extra-lu-mono*)

3.6.4 Finiteness of extrapolations

abbreviation *dbm-default M n* ≡ (∀ *i* > *n*. ∀ *j*. *M i j* = *0*) ∧ (∀ *j* > *n*. ∀ *i*. *M i j* = *0*)

lemma *norm-default-preservation*:
dbm-default M n ⟹ *dbm-default* (*norm M k n*) *n*
by (*simp add*: *norm-def norm-diag-def DBM.neutral dbm-lt.simps*)

lemma *extra-lu-default-preservation*:
dbm-default M n ⟹ *dbm-default* (*extra-lu M L U n*) *n*
by (*simp add*: *extra-lu-def norm-diag-def DBM.neutral dbm-lt.simps*)

instance *int* :: *linordered-cancel-ab-monoid-add* **by** (*standard*; *simp*)

lemmas *finite-subset-rev*[*intro?*] = *finite-subset*[*rotated*]
lemmas [*intro?*] = *finite-subset*

lemma *extra-lu-finite*:
fixes *L U* :: *nat* ⇒ *nat*
shows *finite* {*extra-lu M L U n* | *M*. *dbm-default M n*}
proof −
let *?u* = *Max* {*L i* | *i*. *i* ≤ *n*} **let** *?l* = − *Max* {*U i* | *i*. *i* ≤ *n*}
let *?S* = (*Le* ‘ {*d* :: *int*. *?l* ≤ *d* ∧ *d* ≤ *?u*}) ∪ (*Lt* ‘ {*d* :: *int*. *?l* ≤ *d* ∧ *d* ≤ *?u*}) ∪ {*Le 0*, *Lt 0*, ∞}
from *finite-set-of-finite-funs2*[*of* {*0..n*} {*0..n*} *?S*] **have** *fin*:
finite {*f*. ∀ *x y*. (*x* ∈ {*0..n*} ∧ *y* ∈ {*0..n*} ⟶ *f x y* ∈ *?S*)
∧ (*x* ∉ {*0..n*} ⟶ *f x y* = *0*) ∧ (*y* ∉ {*0..n*} ⟶ *f x y* = *0*)}
(**is** *finite ?R*)
by *auto*
{ fix *M* :: *int DBM* **assume** *A*: *dbm-default M n*
let *?M* = *extra-lu M L U n*
from *extra-lu-default-preservation*[*OF A*] **have** *A*: *dbm-default ?M n* **.**
{ fix *i j* **assume** *i* ∈ {*0..n*} *j* ∈ {*0..n*}

then have *B*: $i \leq n$ $j \leq n$
  by *auto*
have *?M i j* ∈ *?S*
proof (*cases ?M i j* ∈ {*Le 0*, *Lt 0*, ∞})
  case *True* **then show** *?thesis*
    by *auto*
next
  case *F*: *False*
  note *not-inf* = *this*
  have *?l* ≤ *get-const* (*?M i j*) ∧ *get-const* (*?M i j*) ≤ *?u*
  proof (*cases i* = *0*)
    case *True*
    show *?thesis*
    proof (*cases j* = *0*)
      case *True*
      with ‹*i* = *0*› *A F* **show** *?thesis*
        unfolding *extra-lu-def* **by** (*auto simp*: *neutral norm-diag-def*)
    next
      case *False*
       with ‹*i* = *0*› *B not-inf* **have** *?M i j* ≤ *Le 0* *Lt* (−*int* (*U j*)) ≤ *?M i j*
        unfolding *extra-lu-def* **by** (*auto simp*: *Let-def less*[*symmetric*] *intro*: *any-le-inf*)
      with *not-inf* **have** *get-const* (*?M i j*) ≤ *0* −*U j* ≤ *get-const* (*?M i j*)
        by (*cases ?M i j*; *auto*)+
      moreover from ‹*j* ≤ *n*› **have** − *U j* ≥ *?l*
        by (*auto intro*: *Max-ge*)
      ultimately show *?thesis*
        by *auto*
    qed
  next
    case *False*
    then have *i* > *0* **by** *simp*
    show *?thesis*
    proof (*cases j* = *0*)
      case *True*
       with ‹*i* > *0*› *A*(*1*) *B not-inf* **have** *Lt 0* ≤ *?M i j* *?M i j* ≤ *Le* (*int* (*L i*))
        unfolding *extra-lu-def* **by** (*auto simp*: *Let-def less*[*symmetric*] *intro*: *any-le-inf*)
      with *not-inf* **have** *0* ≤ *get-const* (*?M i j*) *get-const* (*?M i j*) ≤ *L i*
        by (*cases ?M i j*; *auto*)+

```
          moreover from ‹i ≤ n› have L i ≤ ?u
            by (auto intro: Max-ge)
          ultimately show ?thesis
            by auto
        next
          case False
          with ‹i > 0› A(1) B not-inf F have
            Lt (−int (U j)) ≤ ?M i j ?M i j ≤ Le (int (L i))
            unfolding extra-lu-def
            by (auto simp: Let-def less[symmetric] neutral norm-diag-def
                  intro: any-le-inf split: if-split-asm)
          with not-inf have − U j ≤ get-const (?M i j) get-const (?M i j)
≤ L i
            by (cases ?M i j; auto)+
          moreover from ‹i ≤ n› ‹j ≤ n› have ?l ≤ − U j L i ≤ ?u
            by (auto intro: Max-ge)
          ultimately show ?thesis
            by auto
        qed
      qed
      then show ?thesis by (cases ?M i j; auto elim: Ints-cases)
    qed
  } moreover
  { fix i j assume i ∉ {0..n}
    with A have ?M i j = 0 by auto
  } moreover
  { fix i j assume j ∉ {0..n}
    with A have ?M i j = 0 by auto
  } moreover note the = calculation
 } then have {extra-lu M L U n | M. dbm-default M n} ⊆ ?R
    by blast
 with fin show ?thesis ..
qed

lemma normalized-integral-dbms-finite:
 finite {norm M (k :: nat ⇒ nat) n | M. dbm-default M n}
 unfolding norm-is-extra by (rule extra-lu-finite)

end
```

4 DBMs as Constraint Systems

theory *DBM-Constraint-Systems*

imports
DBM-Operations
DBM-Normalization
begin

4.1 Misc

lemma *Max-le-MinI*:
assumes *finite S finite T* $S \neq \{\}$ $T \neq \{\}$ $\bigwedge x\ y.\ x \in S \Longrightarrow y \in T \Longrightarrow x \leq y$
shows $Max\ S \leq Min\ T$ **by** (*simp add*: *assms*)

lemma *Min-insert-cases*:
assumes $x = Min$ (*insert a S*) *finite S*
obtains (*default*) $x = a$ | (*elem*) $x \in S$
by (*metis Min-in assms finite.insertI insertE insert-not-empty*)

lemma *cval-add-simp*[*simp*]:
$(u \oplus d)\ x = u\ x + d$
unfolding *cval-add-def* **by** *simp*

lemmas [*simp*] = *any-le-inf*

lemma *Le-in-between*:
assumes $a < b$
obtains d **where** $a \leq Le\ d$ $Le\ d \leq b$
using *assms* **by** *atomize-elim* (*cases a*; *cases b*; *auto*)

lemma *DBMEntry-le-to-sum*:
fixes $e\ e'$ :: *'t* :: *time DBMEntry*
assumes $e' \neq \infty$ $e \leq e'$
shows $-\ e' + e \leq 0$
using *assms* **by** (*cases e*; *cases e'*) (*auto simp*: *DBM.neutral DBM.add uminus*)

lemma *DBMEntry-le-add*:
fixes $a\ b\ c$ :: *'t* :: *time DBMEntry*
assumes $a \leq b + c$ $c \neq \infty$
shows $-c + a \leq b$
using *assms*
by (*cases a*; *cases b*; *cases c*) (*auto simp*: *DBM.neutral DBM.add uminus algebra-simps*)

lemma *DBM-triv-emptyI*:

assumes $M\ 0\ 0 < 0$
shows $[M]_{v,n} = \{\}$
using *assms*
unfolding *DBM-zone-repr-def DBM-val-bounded-def DBM.less-eq*[*symmetric*] *DBM.neutral* **by** *auto*

4.2 Definition and Semantics of Constraint Systems

datatype $('x,\ 'v)$ *constr* =
Lower $'x\ 'v$ *DBMEntry* | *Upper* $'x\ 'v$ *DBMEntry* | *Diff* $'x\ 'x\ 'v$ *DBMEntry*

type-synonym $('x,\ 'v)$ *cs* = $('x,\ 'v)$ *constr set*

inductive *entry-sem* (‹- $\models_e$ -› [62, 62] 62) **where**
$v \models_e Lt\ x$ **if** $v < x$ |
$v \models_e Le\ x$ **if** $v \leq x$ |
$v \models_e \infty$

inductive *constr-sem* (‹- $\models_c$ -› [62, 62] 62) **where**
$u \models_c Lower\ x\ e$ **if** $-\ u\ x \models_e e$ |
$u \models_c Upper\ x\ e$ **if** $u\ x \models_e e$ |
$u \models_c Diff\ x\ y\ e$ **if** $u\ x - u\ y \models_e e$

definition *cs-sem* (‹- $\models_{cs}$ -› [62, 62] 62) **where**
$u \models_{cs} cs \longleftrightarrow (\forall c \in cs.\ u \models_c c)$

definition *cs-models* (‹- $\models$ -› [62, 62] 62) **where**
$cs \models c \equiv \forall u.\ u \models_{cs} cs \longrightarrow u \models_c c$

definition *cs-equiv* (‹- $\equiv_{cs}$ -› [62, 62] 62) **where**
$cs \equiv_{cs} cs' \equiv \forall u.\ u \models_{cs} cs \longleftrightarrow u \models_{cs} cs'$

definition
closure $cs \equiv \{c.\ cs \models c\}$

definition
bot-cs = {*Lower undefined* (*Lt 0*), *Upper undefined* (*Lt 0*)}

lemma *constr-sem-less-eq-iff*:
$u \models_c Lower\ x\ e \longleftrightarrow Le\ (-u\ x) \leq e$
$u \models_c Upper\ x\ e \longleftrightarrow Le\ (u\ x) \leq e$
$u \models_c Diff\ x\ y\ e \longleftrightarrow Le\ (u\ x - u\ y) \leq e$
by (*cases e*; *auto simp*: *constr-sem.simps entry-sem.simps*)+

lemma *constr-sem-mono*:
assumes $e \leq e'$
shows
$u \models_c \textit{Lower } x\ e \implies u \models_c \textit{Lower } x\ e'$
$u \models_c \textit{Upper } x\ e \implies u \models_c \textit{Upper } x\ e'$
$u \models_c \textit{Diff } x\ y\ e \implies u \models_c \textit{Diff } x\ y\ e'$
using *assms* **unfolding** *constr-sem-less-eq-iff* **by** *simp+*

lemma *constr-sem-triv*[*simp*,*intro*]:
$u \models_c \textit{Upper } x\ \infty\ u \models_c \textit{Lower } y\ \infty\ u \models_c \textit{Diff } x\ y\ \infty$
unfolding *constr-sem.simps entry-sem.simps* **by** *auto*

lemma *cs-sem-antimono*:
assumes $cs \subseteq cs'\ u \models_{cs} cs'$
shows $u \models_{cs} cs$
using *assms* **unfolding** *cs-sem-def* **by** *auto*

lemma *cs-equivD*[*intro*, *dest*]:
assumes $u \models_{cs} cs\ cs \equiv_{cs} cs'$
shows $u \models_{cs} cs'$
using *assms* **unfolding** *cs-equiv-def* **by** *auto*

lemma *cs-equiv-sym*:
$cs \equiv_{cs} cs'$ **if** $cs' \equiv_{cs} cs$
using *that* **unfolding** *cs-equiv-def* **by** *fast*

lemma *cs-equiv-union*:
$cs \equiv_{cs} cs \cup cs'$ **if** $cs \equiv_{cs} cs'$
using *that* **unfolding** *cs-equiv-def cs-sem-def* **by** *blast*

lemma *cs-equiv-alt-def*:
$cs \equiv_{cs} cs' \longleftrightarrow (\forall c.\ cs \models c \longleftrightarrow cs' \models c)$
unfolding *cs-equiv-def cs-models-def cs-sem-def* **by** *auto*

lemma *closure-equiv*:
$\textit{closure } cs \equiv_{cs} cs$
unfolding *cs-equiv-alt-def closure-def cs-models-def cs-sem-def* **by** *auto*

lemma *closure-superset*:
$cs \subseteq \textit{closure } cs$
unfolding *closure-def cs-models-def cs-sem-def* **by** *auto*

lemma *bot-cs-empty*:
$\neg\ (u :: ('c \Rightarrow 't :: \textit{linordered-ab-group-add})) \models_{cs} \textit{bot-cs}$

unfolding *bot-cs-def cs-sem-def* **by** (*auto elim*!: *constr-sem.cases entry-sem.cases*)

lemma *finite-bot-cs*:
finite bot-cs
unfolding *bot-cs-def* **by** *auto*

definition *cs-vars* **where**
$cs\text{-}vars\ cs = \bigcup\ (set1\text{-}constr\ `\ cs)$

definition *map-cs-vars* **where**
$map\text{-}cs\text{-}vars\ v\ cs = map\text{-}constr\ v\ id\ `\ cs$

lemma *constr-sem-rename-vars*:
assumes *inj-on v S set1-constr c* $\subseteq$ *S*
shows ($u\ o\ inv\text{-}into\ S\ v) \models_c map\text{-}constr\ v\ id\ c \longleftrightarrow u \models_c c$
using *assms*
by (*cases c*) (*auto intro*!: *constr-sem.intros elim*!: *constr-sem.cases simp*: *DBMEntry.map-id*)

lemma *cs-sem-rename-vars*:
assumes *inj-on v* (*cs-vars cs*)
shows ($u\ o\ inv\text{-}into\ (cs\text{-}vars\ cs)\ v) \models_{cs} map\text{-}cs\text{-}vars\ v\ cs \longleftrightarrow u \models_{cs} cs$
using *assms constr-sem-rename-vars* **unfolding** *map-cs-vars-def cs-sem-def cs-vars-def* **by** *blast*

4.3 Conversion of DBMs to Constraint Systems and Back

definition *dbm-to-cs* :: $nat \Rightarrow ('x \Rightarrow nat) \Rightarrow ('v :: \{linorder,zero\})\ DBM \Rightarrow ('x, 'v)\ cs$ **where**
$dbm\text{-}to\text{-}cs\ n\ v\ M \equiv if\ M\ 0\ 0 < 0\ then\ bot\text{-}cs\ else$
$\{Lower\ x\ (M\ 0\ (v\ x))\ |\ x.\ v\ x \leq n\} \cup$
$\{Upper\ x\ (M\ (v\ x)\ 0)\ |\ x.\ v\ x \leq n\} \cup$
$\{Diff\ x\ y\ (M\ (v\ x)\ (v\ y))\ |\ x\ y.\ v\ x \leq n \wedge v\ y \leq n\}$

lemma *dbm-entry-val-Lower-iff*:
$dbm\text{-}entry\text{-}val\ u\ None\ (Some\ x)\ e \longleftrightarrow u \models_c Lower\ x\ e$
by (*cases e*) (*auto simp*: *constr-sem-less-eq-iff*)

lemma *dbm-entry-val-Upper-iff*:
$dbm\text{-}entry\text{-}val\ u\ (Some\ x)\ None\ e \longleftrightarrow u \models_c Upper\ x\ e$
by (*cases e*) (*auto simp*: *constr-sem-less-eq-iff*)

lemma *dbm-entry-val-Diff-iff*:

dbm-entry-val u (Some x) (Some y) e $\longleftrightarrow$ *u* $\models_c$ *Diff x y e*
by (*cases e*) (*auto simp*: *constr-sem-less-eq-iff*)

lemmas *dbm-entry-val-constr-sem-iff* =
dbm-entry-val-Lower-iff
dbm-entry-val-Upper-iff
dbm-entry-val-Diff-iff

theorem *dbm-to-cs-correct*:
$u \vdash_{v,n} M \longleftrightarrow u \models_{cs}$ *dbm-to-cs n v M*
apply (*rule iffI*)
unfolding *DBM-val-bounded-def dbm-entry-val-constr-sem-iff dbm-to-cs-def*
subgoal
by (*auto simp*: *DBM.neutral DBM.less-eq*[*symmetric*] *cs-sem-def*)
using *bot-cs-empty* **by** (*cases M 0 0 < 0*, *auto simp*: *DBM.neutral DBM.less-eq*[*symmetric*] *cs-sem-def*)

definition
cs-to-dbm v cs ≡ *if* (∀ *u*. ¬*u* $\models_{cs}$ *cs*) *then* (λ- -. *Lt 0*) *else* (
λ*i j*.
if i = *0 then*
if j = *0 then*
Le 0
else
Min (*insert* ∞ {*e*. ∃ *x*. *Lower x e* ∈ *cs* ∧ *v x* = *j*})
else
if j = *0 then*
Min (*insert* ∞ {*e*. ∃ *x*. *Upper x e* ∈ *cs* ∧ *v x* = *i*})
else
Min (*insert* ∞ {*e*. ∃ *x y*. *Diff x y e* ∈ *cs* ∧ *v x* = *i* ∧ *v y* = *j*})
)

lemma *finite-dbm-to-cs*:
assumes *finite* {*x*. *v x* ≤ *n*}
shows *finite* (*dbm-to-cs n v M*)
using [[*simproc add*: *finite-Collect*]] **unfolding** *dbm-to-cs-def*
by (*auto intro*: *assms simp*: *finite-bot-cs*)

lemma *empty-dbm-empty*:
$u \vdash_{v,n}$ (λ- -. *Lt 0*) $\longleftrightarrow$ *False*
unfolding *DBM-val-bounded-def* **by** (*auto simp*: *DBM.less-eq*[*symmetric*])

fun *expr-of-constr* **where**
expr-of-constr (*Lower - e*) = *e* |

```
  expr-of-constr (Upper - e) = e |
  expr-of-constr (Diff - - e) = e

lemma cs-to-dbm1:
  assumes ∀ x ∈ cs-vars cs. v x > 0 ∧ v x ≤ n finite cs
  assumes u ⊢v,n cs-to-dbm v cs
  shows u ⊨cs cs
proof (cases ∀ u. ¬u ⊨cs cs)
  case True
  with assms(3) show ?thesis
    unfolding cs-to-dbm-def by (simp add: empty-dbm-empty)
next
  case False
  show u ⊨cs cs
    unfolding cs-sem-def
  proof (rule ballI)
    fix c
    assume c ∈ cs
    show u ⊨c c
    proof (cases c)
      case (Lower x e)
      with assms(1) ‹c ∈ cs› have ∗: 0 < v x v x ≤ n
        by (auto simp: cs-vars-def)
      let ?S = {e. ∃ x'. Lower x' e ∈ cs ∧ v x' = v x}
      let ?e = Min (insert ∞ ?S)
      have ?S ⊆ expr-of-constr ‘ cs
        by force
      with ‹finite cs› ‹c ∈ cs› ‹c = -› have ?e ≤ e
        using finite-subset finite-imageI by (blast intro: Min-le)
      moreover from ∗ assms(3) False have dbm-entry-val u None (Some
x) ?e
        unfolding DBM-val-bounded-def cs-to-dbm-def by (auto 4 4)
      ultimately have dbm-entry-val u None (Some x) (e)
        by − (rule dbm-entry-val-mono[folded DBM.less-eq])
      then show ?thesis
        unfolding dbm-entry-val-constr-sem-iff[symmetric] ‹c = -› .
    next
      case (Upper x e)
      with assms(1) ‹c ∈ cs› have ∗: 0 < v x v x ≤ n
        by (auto simp: cs-vars-def)
      let ?S = {e. ∃ x'. Upper x' e ∈ cs ∧ v x' = v x}
      let ?e = Min (insert ∞ ?S)
      have ?S ⊆ expr-of-constr ‘ cs
        by force
```

with ‹*finite cs*› ‹$c \in cs$› ‹$c = $ -› **have** $?e \leq e$
using *finite-subset finite-imageI* **by** (*blast intro*: *Min-le*)
moreover from $*$ *assms*(*3*) *False* **have** *dbm-entry-val u* (*Some x*) *None ?e*
unfolding *DBM-val-bounded-def cs-to-dbm-def* **by** (*auto 4 4*)
ultimately have *dbm-entry-val u* (*Some x*) *None e*
by − (*rule dbm-entry-val-mono*[*folded DBM.less-eq*])
then show *?thesis*
unfolding *dbm-entry-val-constr-sem-iff* ‹$c = $ -› **.**
next
case (*Diff x y e*)
with *assms*(*1*) ‹$c \in cs$› **have** $*$: $0 < v\ x$ $v\ x \leq n$ $0 < v\ y$ $v\ y \leq n$
by (*auto simp*: *cs-vars-def*)
let $?S = \{e.\ \exists x'\ y'.\ Diff\ x'\ y'\ e \in cs \wedge v\ x' = v\ x \wedge v\ y' = v\ y\}$
let $?e = Min\ (insert\ \infty\ ?S)$
have $?S \subseteq$ *expr-of-constr* ‘ *cs*
by *force*
with ‹*finite cs*› ‹$c \in cs$› ‹$c = $ -› **have** $?e \leq e$
using *finite-subset finite-imageI* **by** (*blast intro*: *Min-le*)
moreover from $*$ *assms*(*3*) *False* **have** *dbm-entry-val u* (*Some x*) (*Some y*) *?e*
unfolding *DBM-val-bounded-def cs-to-dbm-def* **by** (*auto 4 4*)
ultimately have *dbm-entry-val u* (*Some x*) (*Some y*) *e*
by − (*rule dbm-entry-val-mono*[*folded DBM.less-eq*])
then show *?thesis*
unfolding *dbm-entry-val-constr-sem-iff* ‹$c = $ -› **.**
qed
qed
qed

lemma *cs-to-dbm2*:
assumes $\forall x.\ v\ x \leq n \longrightarrow v\ x > 0$ $\forall x\ y.\ v\ x \leq n \wedge v\ y \leq n \wedge v\ x = v\ y \longrightarrow x = y$
assumes *finite cs*
assumes $u \models_{cs} cs$
shows $u \vdash_{v,n}$ *cs-to-dbm v cs*
proof (*cases* $\forall u.\ \neg u \models_{cs} cs$)
case *True*
with *assms* **show** *?thesis*
unfolding *cs-to-dbm-def* **by** (*simp add*: *empty-dbm-empty*)
next
case *False*
let $?M =$ *cs-to-dbm v cs*
show $u \vdash_{v,n}$ *cs-to-dbm v cs*

```
    unfolding DBM-val-bounded-def DBM.less-eq[symmetric]
  proof (safe)
    show Le 0 ≤ cs-to-dbm v cs 0 0
      using False unfolding cs-to-dbm-def by auto
  next
    fix x :: 'a
    assume v x ≤ n
    let ?S = {e. ∃x'. Lower x' e ∈ cs ∧ v x' = v x}
    from ‹v x ≤ n› assms have v x > 0
      by simp
    with False have ?M 0 (v x) = Min (insert ∞ ?S)
      unfolding cs-to-dbm-def by auto
    moreover have finite ?S
    proof -
      have ?S ⊆ expr-of-constr ` cs
        by force
      also have finite ...
        using ‹finite cs› by (rule finite-imageI)
      finally show ?thesis .
    qed
    ultimately show dbm-entry-val u None (Some x) (?M 0 (v x))
      using assms(2-) ‹v x ≤ n›
      apply (cases rule: Min-insert-cases)
       apply auto[]
      apply (simp add: dbm-entry-val-constr-sem-iff cs-sem-def, metis)
      done
  next
    fix x :: 'a
    assume v x ≤ n
    let ?S = {e. ∃x'. Upper x' e ∈ cs ∧ v x' = v x}
    from ‹v x ≤ n› assms have v x > 0
      by simp
    with False have ?M (v x) 0 = Min (insert ∞ ?S)
      unfolding cs-to-dbm-def by auto
    moreover have finite ?S
    proof -
      have ?S ⊆ expr-of-constr ` cs
        by force
      also have finite ...
        using ‹finite cs› by (rule finite-imageI)
      finally show ?thesis .
    qed
   ultimately show dbm-entry-val u (Some x) None (cs-to-dbm v cs (v x)
0)
```

using ‹$v\ x \le n$› *assms*(*2*−)
apply (*cases rule*: *Min-insert-cases*)
apply *auto*[]
apply (*simp add*: *dbm-entry-val-constr-sem-iff cs-sem-def*, *metis*)
done
next
fix $x\ y$ :: ′*a*
assume $v\ x \le n\ v\ y \le n$
let $?S = \{e.\ \exists\, x'\ y'.\ Diff\ x'\ y'\ e \in cs \wedge v\ x' = v\ x \wedge v\ y' = v\ y\}$
from ‹$v\ x \le n$› ‹$v\ y \le n$› *assms* **have** $v\ x > 0\ v\ y > 0$
by *auto*
with *False* **have** $?M\ (v\ x)\ (v\ y) = Min\ (insert\ \infty\ ?S)$
unfolding *cs-to-dbm-def* **by** *auto*
moreover have *finite ?S*
proof −
have $?S \subseteq$ *expr-of-constr* ‘ *cs*
by *force*
also have *finite* ...
using ‹*finite cs*› **by** (*rule finite-imageI*)
finally show *?thesis* **.**
qed
ultimately show *dbm-entry-val u* (*Some x*) (*Some y*) (*cs-to-dbm v cs* ($v\ x$) ($v\ y$))
using ‹$v\ x \le n$› ‹$v\ y \le n$› *assms*(*2*−)
apply (*cases rule*: *Min-insert-cases*)
apply *auto*[]
apply (*simp add*: *dbm-entry-val-constr-sem-iff cs-sem-def*, *metis*)
done
qed
qed

theorem *cs-to-dbm-correct*:
assumes $\forall\, x \in cs\text{-}vars\ cs.\ v\ x \le n\ \forall\, x.\ v\ x \le n \longrightarrow v\ x > 0$
$\forall\, x\ y.\ v\ x \le n \wedge v\ y \le n \wedge v\ x = v\ y \longrightarrow x = y$
finite cs
shows $u \vdash_{v,n} cs\text{-}to\text{-}dbm\ v\ cs \longleftrightarrow u \models_{cs} cs$
using *assms* **by** (*blast intro*: *cs-to-dbm1 cs-to-dbm2*)

corollary *cs-to-dbm-correct*′:
assumes
bij-betw v (*cs-vars cs*) {*1..n*} $\forall\, x.\ v\ x \le n \longrightarrow v\ x > 0\ \forall\, x.\ x \notin cs\text{-}vars\ cs \longrightarrow v\ x > n$
finite cs
shows $u \vdash_{v,n} cs\text{-}to\text{-}dbm\ v\ cs \longleftrightarrow u \models_{cs} cs$

proof (*rule cs-to-dbm-correct* , *safe*)
  fix *x* **assume** $x \in$ *cs-vars cs*
  then show $v\ x \leq n$
    using *assms*(*1*) **unfolding** *bij-betw-def* **by** *auto*
next
  fix *x* **assume** $v\ x \leq n$
  then show $0 < v\ x$
    using *assms*(*2*) **by** *blast*
next
  fix *x y* :: *'a*
  assume *A*: $v\ x \leq n\ v\ y \leq n\ v\ x = v\ y$
  with *A assms* **show** $x = y$
    unfolding *bij-betw-def* **by** (*auto dest*!: *inj-onD*)
next
  show *finite cs*
    by (*rule assms*)
qed

4.4 Application: Relaxation On Constraint Systems

The following is a sample application of viewing DBMs as constraint systems. We show define an equivalent of the *up* operation on DBMs, prove it correct, and then derive an alternative correctness proof for *up*.

definition
  up-cs cs = {*c*. $c \in cs \wedge$ (*case c of Upper* - - ⇒ *False* | - ⇒ *True*)}

lemma *Lower-shiftI*:
  $u \oplus d \models_c$ *Lower x e* **if** $u \models_c$ *Lower x e* (*d* :: *'t* :: *linordered-ab-group-add*) ≥ 0
  using *that diff-mono less-trans not-less-iff-gr-or-eq*
  by (*cases e*;*fastforce simp*: *constr-sem-less-eq-iff*)

lemma *Upper-shiftI*:
  $u \oplus d \models_c$ *Upper x e* **if** $u \models_c$ *Upper x e* (*d* :: *'t* :: *linordered-ab-group-add*) ≤ 0
  using *that add-less-le-mono*
  by (*cases e*) (*fastforce simp*: *constr-sem-less-eq-iff add.commute add-decreasing*)+

lemma *Diff-shift*:
  $u \oplus d \models_c$ *Diff x y e* $\longleftrightarrow u \models_c$ *Diff x y e* **for** *d* :: *'t* :: *linordered-ab-group-add*
  by (*cases e*) (*auto simp*: *constr-sem-less-eq-iff*)

lemma *up-cs-complete*:

$u \oplus d \models_{cs}$ *up-cs cs* **if** $u \models_{cs}$ *cs* $d \geq 0$ **for** d :: *'t* :: *linordered-ab-group-add*
using *that* **unfolding** *up-cs-def cs-sem-def*
apply *clarsimp*
subgoal for *x*
by (*cases x*) (*auto simp*: *Diff-shift intro*: *Lower-shiftI*)
done

definition
lower-upper-closed cs $\equiv \forall$ *x y e e'*.
Lower x e $\in$ *cs* $\wedge$ *Upper y e'* $\in$ *cs* $\longrightarrow$ ($\exists$ *e''*. *Diff y x e''* $\in$ *cs* $\wedge$ *e''* $\leq$ *e* + *e'*)

lemma *up-cs-sound*:
assumes $u \models_{cs}$ *up-cs cs lower-upper-closed cs finite cs*
obtains *u'* **and** *d* :: *'t* :: *time* **where** $d \geq 0$ $u' \models_{cs}$ *cs* $u = u' \oplus d$
proof −
define *U* **and** *L* **and** *LT* **where**
$U \equiv \{e + Le\ (-u\ x) \mid x\ e.\ Upper\ x\ e \in cs \wedge e \neq \infty\}$
and $L \equiv \{-e + Le\ (-u\ x) \mid x\ e.\ Lower\ x\ e \in cs \wedge e \neq \infty\}$
and $LT \equiv \{Le\ (-d - u\ x) \mid x\ d.\ Lower\ x\ (Lt\ d) \in cs\}$
note *defs* = *U-def L-def LT-def*
let *?l* = *Max L* **and** *?u* = *Min U*
have $LT \subseteq L$
by (*force simp*: *DBM-arith-defs defs*)
have *Diff-semD*: $u \models_c$ *Diff y x* (*e* + *e'*) **if** *Lower x e* $\in$ *cs Upper y e'* $\in$ *cs* **for** *x y e e'*
proof −
from *assms that* **obtain** *e''* **where** *Diff y x e''* $\in$ *cs e''* $\leq$ *e* + *e'*
unfolding *lower-upper-closed-def cs-equiv-def* **by** *blast*
with *assms*(*1*) **show** *?thesis*
unfolding *cs-sem-def up-cs-def* **by** (*auto intro*: *constr-sem-mono*)
qed
have *Lower-semD*: $u \models_c$ *Lower x e* **if** *Lower x e* $\in$ *cs* **for** *x e*
using *that assms* **unfolding** *cs-sem-def up-cs-def* **by** *auto*
have *Lower-boundI*: $-e + Le\ (-u\ x) \leq 0$ **if** *Lower x e* $\in$ *cs* $e \neq \infty$ **for** *x e*
using *Lower-semD*[*OF that*(*1*)] *that*(*2*) **unfolding** *constr-sem-less-eq-iff*
by (*intro DBMEntry-le-to-sum*)
from ‹*finite cs*› **have** *finite L*
unfolding *defs*
by (*force intro*: *finite-subset*[**where** *B* = (λ*c*. *case c of Lower x e* $\Rightarrow$ $-e + Le\ (-\ u\ x)$) ‘ *cs*])
from ‹*finite cs*› **have** *finite U*

unfolding *defs*
by (*force intro*: *finite-subset*[**where** $B = (\lambda c.\ case\ c\ of\ Upper\ x\ e \Rightarrow e + Le\ (-\ u\ x))$ ‘ *cs*])
note *L-ge* = *Max-ge*[*OF* ‹*finite L*›] **and** *U-le* = *Min-le*[*OF* ‹*finite U*›]
have *L-0*: $Max\ L \leq 0$ **if** $L \neq \{\}$
by (*intro Max.boundedI* ‹*finite L*› *that*) (*auto intro*: *Lower-boundI simp*: *defs*)
have *L-U*: $Max\ L \leq Min\ U$ **if** $L \neq \{\}$ $U \neq \{\}$
apply (*intro Max-le-MinI* ‹*finite L*› ‹*finite U*› *that*)
apply (*clarsimp simp*: *defs*)
apply (*drule* (*1*) *Diff-semD*)
subgoal for $x\ y\ e\ e'$
unfolding *constr-sem-less-eq-iff*
by (*cases e*; *cases* e'; *simp add*: *DBM-arith-defs*; *simp add*: *algebra-simps*)
done
consider
(*L-empty*) $L = \{\}$ | (*Lt-empty*) $LT = \{\}$ | (*L-gt-Lt*) $Max\ L > Max\ LT$ |
(*Lt-Max*) *x d* **where** *Lower x* (*Lt d*) $\in$ *cs Le* $(-d\ -\ u\ x) \in LT$ *Max L* $= Le\ (-d\ -\ u\ x)$
by (*smt* (*verit*) *finite-subset Max-in Max-mono* ‹*finite L*› ‹$LT \subseteq L$› *less-le mem-Collect-eq defs*)
note *L-Lt-cases* = *this*
have *Lt-Max-rule*: $-\ c\ -\ u\ x < 0$
if *Lower x* (*Lt c*) $\in$ *cs Max L* $= Le\ (-\ c\ -\ u\ x)$ $L \neq \{\}$ **for** *c x*
using *that*
by (*metis DBMEntry.distinct*(*1*) *L-0 Le-le-LeD Le-less-Lt Lower-semD add.inverse-inverse constr-sem-less-eq-iff*(*1*) *eq-iff-diff-eq-0 less-le neutral*)
have *LT-0-boundI*: $\exists\, d \leq 0.\ (\forall\, l \in L.\ l \leq Le\ d) \wedge (\forall\, l \in LT.\ l < Le\ d)$ **if** ‹$L \neq \{\}$›
proof −
obtain *d* **where** *d*: $?l \leq Le\ d$ $d \leq 0$
by (*metis L-0* ‹$L \neq \{\}$› *neutral order-refl*)
show *?thesis*
proof (*cases rule*: *L-Lt-cases*)
case *L-empty*
with ‹$L \neq \{\}$› **show** *?thesis*
by *simp*
next
case *Lt-empty*
then show *?thesis*
by (*smt* (*verit*) *L-ge d*(*1*,*2*) *empty-iff leD leI less-le-trans*)
next

case *L-gt-Lt*
then show *?thesis*
by (*smt* (*verit*) *finite-subset Max-ge* ‹*finite L*› ‹$LT \subseteq L$› *d*(*1*,*2*) *leD leI less-le-trans*)
next
case (*Lt-Max x c*)
define *d* **where** $d \equiv - c - u\ x$
from *Lt-Max*(*1*,*3*) ‹$L \neq \{\}$› **have** $d < 0$
unfolding *d-def* **by** (*rule Lt-Max-rule*)
then obtain d' **where** d': $d < d'$ $d' < 0$
using *dense* **by** *auto*
have $\forall l \in L.\ l < Le\ d'$
proof *safe*
fix *l*
assume $l \in L$
then have $l \leq Le\ d$
unfolding *d-def* ‹*Max L* = -›[*symmetric*] **by** (*rule L-ge*)
also from d' **have** $\ldots < Le\ d'$
by *auto*
finally show $l < Le\ d'$ **.**
qed
with *Lt-Max*(*1*,*3*) d' ‹*finite L*› ‹$L \neq \{\}$› ‹$LT \subseteq L$› **show** *?thesis*
by (*intro exI*[*of - d'*]) *auto*
qed
qed
consider
(*none*) $L = \{\}$ $U = \{\}$
| (*upper*) $L = \{\}$ $U \neq \{\}$
| (*lower*) $L \neq \{\}$ $U = \{\}$
| (*proper*) $L \neq \{\}$ $U \neq \{\}$
by *force*

The main statement of of the proof. Note that most of the lengthiness of the proof is owed to the third conjunct. Our initial hope was that this conjunct would not be needed.

then obtain *d* **where** *d*: $d \leq 0$ $\forall l \in L.\ l \leq Le\ d$ $\forall l \in LT.\ l < Le\ d$ $\forall u \in U.\ Le\ d \leq u$
proof *cases*
case *none*
then show *?thesis*
by (*intro that*[*of 0*]) (*auto simp*: *defs*)
next
case *upper*
obtain *d* **where** $Le\ d \leq Min\ U$ $d \leq 0$

```
    by (smt (verit) DBMEntry.distinct(3) add-inf(2) any-le-inf neg-le-0-iff-le
DBM.neutral
          order.not-eq-order-implies-strict sum-gt-neutral-dest')
    then show ?thesis
      using upper ‹finite U› by (intro that[of d]) (auto simp: defs)
  next
    case lower
    obtain d where d: Max L ≤ Le d d ≤ 0
      by (smt (verit) L-0 lower(1) neutral order-refl)
    show ?thesis
    proof (cases rule: L-Lt-cases)
      case L-empty
      with lower(1) show ?thesis
        by simp
    next
      case Lt-empty
      then show ?thesis
       by (metis (lifting) L-ge d(1,2) empty-iff leD leI less-le-trans lower(2)
that)
    next
      case L-gt-Lt
      then show ?thesis
        using LT-0-boundI lower(1,2) that by blast
    next
      case (Lt-Max x c)
      define d where d ≡ − c − u x
      from Lt-Max(1,3) lower(1) have d < 0
        unfolding d-def by (rule Lt-Max-rule)
      then obtain d' where d': d < d' d' < 0
        using dense by auto
      have ∀ l ∈ L. l < Le d'
      proof safe
        fix l
        assume l ∈ L
        then have l ≤ Le d
          unfolding d-def ‹Max L = -›[symmetric] by (rule L-ge)
        also from d' have ... < Le d'
          by auto
        finally show l < Le d' .
      qed
      with Lt-Max(1,3) d' ‹finite L› lower ‹LT ⊆ L› show ?thesis
        by (intro that[of d']) auto
    qed
  next
```

case *proper*
with *L-U L-0* **have** *Max L* $\leq$ *Min U Max L* $\leq$ *0*
by *auto*
from ‹*finite U*› ‹$U \neq \{\}$› **have** $?u \in U$
unfolding *U-def* **by** (*rule Min-in*)
have *main*:
$\exists\, d'.\ -d - u\ x < d' \wedge Le\ d' < ?u$
if $Lower\ x\ (Lt\ d) \in cs$ $Le\ (-d - u\ x) \in LT$ $?l = Le\ (-d - u\ x)$ **for** *d x*
proof (*cases ?u*)
case ($Le\ d'$)
with ‹$?u \in U$› **obtain** *e y* **where** $*$: $Le\ d' = e + Le\ (-\ u\ y)$ $Upper\ y\ e \in cs$
unfolding *U-def* **by** *auto*
then obtain *d1* **where** $e = Le\ d1$
by (*cases e*) (*auto simp*: *DBM-arith-defs*)
with $*$ **have** $d' = d1 - u\ y$
by (*auto simp*: *DBM-arith-defs*)
from *Diff-semD*[*OF* ‹$Lower\ x\ (Lt\ d) \in cs$› ‹$Upper\ y\ e \in$ -›] **have** $u\ y - u\ x < d + d1$
unfolding *constr-sem-less-eq-iff* ‹$e =$ -› **by** (*simp add*: *DBM-arith-defs*)
then have $-\ d - u\ x < d'$
unfolding ‹$d' =$ -› **by** (*simp add*: *algebra-simps*)
then obtain *d1* **where** $-d - u\ x < d1$ $d1 < d'$
using *dense* **by** *auto*
with ‹$?u =$ -› **show** *?thesis*
by (*intro exI*[**where** $x = d1$]) *auto*
next
case ($Lt\ d'$)
with ‹$?u \in U$› **obtain** *e y* **where** $*$: $Lt\ d' = e + Le\ (-\ u\ y)$ $Upper\ y\ e \in cs$
unfolding *U-def* **by** *auto*
then obtain *d1* **where** $e = Lt\ d1$
by (*cases e*) (*auto simp*: *DBM-arith-defs*)
with $*$ **have** $d' = d1 - u\ y$
by (*auto simp*: *DBM-arith-defs*)
from *Diff-semD*[*OF* ‹$Lower\ x\ (Lt\ d) \in cs$› ‹$Upper\ y\ e \in$ -›] **have** $u\ y - u\ x < d + d1$
unfolding *constr-sem-less-eq-iff* ‹$e =$ -› **by** (*simp add*: *DBM-arith-defs*)
then have $-\ d - u\ x < d'$
unfolding ‹$d' =$ -› **by** (*simp add*: *algebra-simps*)
then obtain *d1* **where** $-d - u\ x < d1$ $d1 < d'$
using *dense* **by** *auto*
with ‹$?u =$ -› **show** *?thesis*

by (*intro exI*[**where** $x = d1$]) *auto*
next
case *INF*
with ‹*?u* ∈ *U*› **show** *?thesis*
using *Lt-Max-rule proper*(*1*) *that*(*1*,*3*) **by** *fastforce*
qed
consider (*eq*) *Max L* = *Min U* | (*0*) *Min U* ≥ *0* | (*gt*) *Max L* < *Min U Min U* < *0*
using ‹*Max L* ≤ *Min U*› **by** *fastforce*
then show *?thesis*
proof *cases*
case *eq*
from *proper* ‹*finite L*› ‹*finite U*› **have** *?l* ∈ *L ?u* ∈ *U*
by − (*rule Max-in Min-in*; *assumption*)+
then obtain *x y e e′* **where** ∗:
?l = − *e* + *Le* (− *u x*) *Lower x e* ∈ *cs e* ≠ ∞
?u = *e′* + *Le* (− *u y*) *Upper y e′* ∈ *cs e′* ≠ ∞
unfolding *defs* **by** *auto*
with ‹*?l* = *?u*› **obtain** *d* **where** *d*: *?l* = *Le d*
apply (*cases e*; *cases e′*; *simp add*: *DBM-arith-defs*)
subgoal for *a b*
proof −
assume *prems*: − *a* − *u x* = *b* − *u y e* = *Le a e′* = *Lt b*
from ∗ **have** *u* $\models_c$ *Diff y x* (*e* + *e′*)
by (*intro Diff-semD*)
with *prems* **have** *False*
by (*simp add*: *DBM-arith-defs constr-sem-less-eq-iff algebra-simps*)
then show *?thesis* **..**
qed
done
from ‹*?l* ≤ *0*› **have** ∗∗: *d* ≤ *0* ∀ *l* ∈ *L*. *l* ≤ *Le d* ∀ *u* ∈ *U*. *Le d* ≤ *u*
apply (*simp add*: *DBM.neutral d*)
apply (*auto simp*: *d*[*symmetric*] *intro*: *L-ge*)[]
apply (*auto simp*: *d*[*symmetric*] *eq intro*: *U-le L-ge*)[]
done
show *?thesis*
proof (*cases rule*: *L-Lt-cases*)
case *L-empty*
with ‹*L* ≠ {}› **show** *?thesis*
by *simp*
next
case *Lt-empty*
with ∗∗ **show** *?thesis*
by (*intro that*[*of d*]) *auto*

next
case *L-gt-Lt*
with ∗∗ **show** *?thesis*
by (*intro that*[*of d*]; *simp*)
(*metis finite-subset Max-ge* ‹*LT* ⊆ *L*› ‹*finite L*› *d le-less-trans*)
next
case (*Lt-Max y d1*)
from *main*[*OF this*] **obtain** *d′* **where** *d′* > − *d1* − *u y Le d′* < *Min U*
by *auto*
with ∗∗ *Lt-Max*(*3*)[*symmetric*] *d eq* **show** *?thesis*
by (*intro that*[*of d′*]; *simp*)
qed
next
case *0*
from *LT-0-boundI*[*OF* ‹*L* ≠ {}›] **obtain** *d* **where** *d* ≤ *0* ∀ *l*∈*L*. *l* ≤ *Le d* ∀ *l*∈*LT*. *l* < *Le d*
by *safe*
with ‹*Max L* ≤ *0*› ‹*finite L*› ‹*finite U*› *proper 0* **show** *?thesis*
by (*intro that*[*of d*]) (*auto simp*: *DBM.neutral intro*: *order-trans*)
next
case *gt*
then obtain *d* **where** *d*: *Max L* ≤ *Le d Le d* ≤ *Min U*
by (*elim Le-in-between*)
with ‹- < *0*› **have** *Le d* < *0*
by *auto*
then have *d* ≤ *0*
by (*simp add*: *neutral*)
show *?thesis*
proof (*cases rule*: *L-Lt-cases*)
case *L-empty*
with ‹*L* ≠ {}› **show** *?thesis*
by *simp*
next
case *Lt-empty*
with *d* ‹*d* ≤ *0*› **show** *?thesis*
using *proper* ‹*finite L*› ‹*finite U*› **by** (*intro that*[*of d*]) (*auto intro*: *L-ge U-le*)
next
case *L-gt-Lt*
with *d* ‹*d* ≤ *0*› *proper* ‹*finite L*› ‹*finite U*› **show** *?thesis*
apply (*intro that*[*of d*])
apply (*auto intro*: *L-ge U-le*)[*2*]
apply (*meson finite-subset Max-ge* ‹*LT* ⊆ *L*› *le-less-trans*

less-le-trans)
apply *simp*
done
next
case (*Lt-Max y d1*)
from *main*[*OF this*] **obtain** d' **where** d': $d' > - d1 - u\ y$ *Le* $d' <$ *Min U*
by *auto*
with *d* **have** *d-bounds*: *?l* < *Le* d' *Le* $d' \leq$ *?u*
unfolding ‹*?l* = -› **by** *auto*
from ‹*?l* < *Le* d'› **have** ∀ *l* ∈ *L*. *l* < *Le* d'
using *Max-less-iff* ‹*finite L*› **by** *blast*
moreover from ‹*Le* $d' \leq$ *?u*› ‹*?u* < *0*› **have** $d' \leq 0$
by (*metis Le-le-LeD le-less-trans neutral order.strict-iff-order*)
with *d Lt-Max*(*3*)[*symmetric*] *d-bounds* d' ‹*LT* ⊆ *L*› **show** *?thesis*
using *proper* ‹*finite L*› ‹*finite U*›
by (*intro that*[*of* d']; *auto*)
qed
qed
qed
have $u \oplus d \models_{cs} cs$
unfolding *cs-sem-def*
proof *safe*
fix *c* :: ('*a*, '*t*) *constr*
assume *c* ∈ *cs*
show $u \oplus d \models_{c} c$
proof (*cases c*)
case (*Lower x e*)
show *?thesis*
proof (*cases* $e = \infty$)
case *True*
with ‹*c* = -› **show** *?thesis*
by (*auto simp*: *constr-sem-less-eq-iff*)
next
case *False*
with ‹*c* = -› ‹*c* ∈ -› **have** $-e +$ *Le* $(-u\ x) \in L$
unfolding *defs* **by** *auto*
with *d* **have** $-e +$ *Le* $(-u\ x) \leq$ *Le d*
by *auto*
then show *?thesis*
using *d*(*3*) ‹*c* ∈ -› **unfolding** ‹*c* = -› *constr-sem-less-eq-iff*
apply (*cases e*; *simp add*: *defs DBM-arith-defs*)
apply (*metis diff-le-eq minus-add-distrib minus-le-iff uminus-add-conv-diff*)
apply (*metis ab-group-add-class.ab-diff-conv-add-uminus leD le-less

less-diff-eq
        minus-diff-eq neg-less-iff-less)
      done
    qed
  next
    case (*Upper x e*)
    show *?thesis*
    proof (*cases e* = ∞)
      case *True*
      with ‹*c* = -› **show** *?thesis*
        by (*auto simp*: *constr-sem-less-eq-iff*)
    next
      case *False*
      with ‹*c* = -› ‹*c* ∈ -› **have** *e* + *Le* (−*u x*) ∈ *U*
        by (*auto simp*: *defs*)
      with *d* **show** *?thesis*
      by (*cases e*) (*auto simp*: ‹*c* = -› *constr-sem-less-eq-iff DBM-arith-defs algebra-simps*)
    qed
  next
    case (*Diff x y e*)
    with *assms* ‹*c* ∈ *cs*› **show** *?thesis*
      by (*auto simp*: *Diff-shift cs-sem-def up-cs-def*)
  qed
qed
with ‹*d* ≤ *0*› **show** *?thesis*
  by (*intro that*[*of* −*d u* ⊕ *d*]; *simp add*: *cval-add-def*)
qed

Note that if we compare this proof to $[\![\forall c.\ 0 < ?v\ c \wedge (\forall x\ y.\ ?v\ x \le ?n \wedge ?v\ y \le ?n \wedge ?v\ x = ?v\ y \longrightarrow x = y);\ ?u \in [up\ ?M]_{?v,?n}]\!] \Longrightarrow ?u \in [?M]_{?v,?n}{}^{\uparrow}$, we can see that we have not gained much. Settling on DBM entry arithmetic as done above was not the optimal choice for this proof, while it can drastically simply some other proofs. Also, note that the final theorem we obtain below (*DBM-up-correct*) is slightly stronger than what we would get with $[\![\forall c.\ 0 < ?v\ c \wedge (\forall x\ y.\ ?v\ x \le ?n \wedge ?v\ y \le ?n \wedge ?v\ x = ?v\ y \longrightarrow x = y);\ ?u \in [up\ ?M]_{?v,?n}]\!] \Longrightarrow ?u \in [?M]_{?v,?n}{}^{\uparrow}$. Finally, note that a more elegant definition of *lower-upper-closed* would probably be: *definition lower-upper-closed cs* ≡ $\forall x\ y\ e\ e'.\ cs \models Lower\ x\ e \wedge cs \models Upper\ y\ e' \longrightarrow (\exists\ e''.\ cs \models Diff\ y\ x\ e'' \wedge e'' \le e + e')$ This would mean that in the proof we would have to replace minimum and maximum by supremum and infimum. The advantage would be that the finiteness assumption could be removed. However, as our DBM entries do not come with −∞, they do not form a complete lattice. Thus we would either have to make this extension or directly refer to the embedded values directly, which would again have to form a complete lattice. Both variants come with some technical inconvenience.

lemma *up-cs-sem*:
  fixes *cs* :: ('*x*, '*v* :: *time*) *cs*
  assumes *lower-upper-closed cs finite cs*
  shows $\{u.\ u \models_{cs}$ *up-cs cs*$\} = \{u \oplus d \mid u\ d.\ u \models_{cs} cs \wedge d \geq 0\}$
  by *safe* (*metis up-cs-sound up-cs-complete assms*)+

definition
  close-lu :: ('*t*::*linordered-cancel-ab-semigroup-add*) *DBM* ⇒ '*t DBM*
where
  close-lu M ≡ λ*i j. if i* > *0 then min* (*dbm-add* (*M i 0*) (*M 0 j*)) (*M i j*) *else M i j*

definition
  up' :: ('*t*::*linordered-cancel-ab-semigroup-add*) *DBM* ⇒ '*t DBM*
where
  up' M ≡ λ*i j. if i* > *0* ∧ *j* = *0 then* ∞ *else M i j*

lemma *up-alt-def*:
  up M = *up'* (*close-lu M*)
  by (*intro ext*) (*simp add*: *up-def up'-def close-lu-def*)

lemma *close-lu-equiv*:
  fixes *M* :: '*t* ::*time DBM*
  shows *dbm-to-cs n v M* $\equiv_{cs}$ *dbm-to-cs n v* (*close-lu M*)
  unfolding *cs-equiv-def dbm-to-cs-correct*[*symmetric*]
    DBM-val-bounded-def close-lu-def dbm-entry-val-constr-sem-iff
  unfolding *min-def DBM.add*[*symmetric*]
  unfolding *constr-sem-less-eq-iff*
  unfolding *DBM.less-eq*[*symmetric*] *DBM.neutral*[*symmetric*]
  apply (*auto simp*:)[]
          apply (*force simp add*: *add-increasing2*)
        apply (*metis* (*full-types*) *le0*)+
  subgoal premises *prems* **for** *u c1 c2*
  proof −
    have *Le* (*u c1* − *u c2*) = *Le* (*u c1*) + *Le* (− *u c2*)
      by (*simp add*: *DBM-arith-defs*)
    also from *prems* **have** ... ≤ *M* (*v c1*) *0* + *M 0* (*v c2*)
      by (*intro add-mono*) *auto*
    finally show *?thesis* **.**
  qed

by (*smt* (*verit*) *leI le-zero-eq order-trans* | *metis le0*)+

lemma *close-lu-closed*:
lower-upper-closed (*dbm-to-cs n v* (*close-lu M*)) **if** $M\ 0\ 0 \geq 0$
using *that* **unfolding** *lower-upper-closed-def dbm-to-cs-def close-lu-def*
apply (*clarsimp*; *safe*)
subgoal
by *auto*
subgoal for *x y*
by (*auto simp*: *DBM.add*[*symmetric*])
(*metis add.commute add.right-neutral add-left-mono min.absorb2 min.cobounded1*)
by (*simp add*: *add-increasing2*)

lemma *close-lu-closed′*: — Unused
lower-upper-closed (*dbm-to-cs n v* (*close-lu M*) ∪ *dbm-to-cs n v M*) **if** $M\ 0\ 0 \geq 0$
using *that* **unfolding** *lower-upper-closed-def dbm-to-cs-def close-lu-def*
apply (*clarsimp*; *safe*)
subgoal
by *auto*
subgoal for *x y*
by (*metis DBM.add add.commute add.right-neutral add-left-mono min.absorb2 min.cobounded1*)
subgoal for *x y*
by (*metis DBM.add add.commute min.cobounded1*)
by (*simp add*: *add-increasing2*)

lemma *up-cs-up′-equiv*:
fixes *M* :: ′*t* ::*time DBM*
assumes $M\ 0\ 0 \geq 0$ *clock-numbering v*
shows *up-cs* (*dbm-to-cs n v M*) $\equiv_{cs}$ *dbm-to-cs n v* (*up′ M*)
using *assms*
unfolding *up′-def up-cs-def cs-equiv-def dbm-to-cs-correct*[*symmetric*]
DBM-val-bounded-def close-lu-def dbm-entry-val-constr-sem-iff
by (*auto split*: *if-split-asm*
simp: *dbm-to-cs-def cs-sem-def DBM.add*[*symmetric*] *DBM.less-eq*[*symmetric*] *DBM.neutral*)

lemma *up-equiv-cong*: — Unused
fixes *cs cs′* :: (′*x*, ′*v* :: *time*) *cs*
assumes *cs* $\equiv_{cs}$ *cs′ finite cs finite cs′ lower-upper-closed cs lower-upper-closed cs′*
shows *up-cs cs* $\equiv_{cs}$ *up-cs cs′*

using *assms* **unfolding** *cs-equiv-def* **by** (*metis up-cs-complete up-cs-sound*)

lemma *DBM-up-correct*:
fixes M :: $'t$ ::*time DBM*
assumes *clock-numbering v finite* $\{x.\ v\ x \leq n\}$
shows $u \in ([M]_{v,n})^{\uparrow} \longleftrightarrow u \in [up\ M]_{v,n}$
proof (*cases M 0 0* $\geq$ *0*)
case *True*
have $u \in ([M]_{v,n})^{\uparrow} \longleftrightarrow (\exists\, d\ u'.\ u' \vdash_{v,n} M \wedge d \geq 0 \wedge u = u' \oplus d)$
unfolding *DBM-zone-repr-def zone-delay-def* **by** *auto*
also have ... $\longleftrightarrow (\exists\, d\ u'.\ u' \models_{cs}$ *dbm-to-cs n v M* $\wedge\ d \geq 0 \wedge u = u' \oplus d)$
unfolding *dbm-to-cs-correct* **..**
also have ... $\longleftrightarrow (\exists\, d\ u'.\ u' \models_{cs}$ *dbm-to-cs n v* (*close-lu M*) $\wedge\ d \geq 0 \wedge u = u' \oplus d)$
using *cs-equivD close-lu-equiv cs-equiv-sym* **by** *metis*
also have ... $\longleftrightarrow u \models_{cs}$ *up-cs* (*dbm-to-cs n v* (*close-lu M*))
proof −
let *?cs* = *dbm-to-cs n v* (*close-lu M*)
have *lower-upper-closed ?cs*
by (*intro close-lu-closed True*)
moreover have *finite ?cs*
by (*intro finite-dbm-to-cs assms*)
ultimately have $\{u.\ u \models_{cs}$ *up-cs ?cs*$\} = \{u \oplus d\ |u\ d.\ u \models_{cs}$ *?cs* $\wedge$ *0* $\leq d\}$
by (*rule up-cs-sem*)
then show *?thesis*
by (*auto 4 3*)
qed
also have ... $\longleftrightarrow u \models_{cs}$ *dbm-to-cs n v* (*up'* (*close-lu M*))
proof −
from ‹*M 0 0* $\geq$ *0*› **have** *up-cs* (*dbm-to-cs n v* (*close-lu M*)) $\equiv_{cs}$ *dbm-to-cs n v* (*up'* (*close-lu M*))
by (*intro up-cs-up'-equiv*[*OF* - ‹*clock-numbering v*›], *simp add*: *close-lu-def*)
then show *?thesis*
using *cs-equivD cs-equiv-sym* **by** *metis*
qed
also have ... $\longleftrightarrow u \models_{cs}$ *dbm-to-cs n v* (*up M*)
unfolding *up-alt-def* **..**
also have ... $\longleftrightarrow u \vdash_{v,n}$ *up M*
unfolding *dbm-to-cs-correct* **..**
also have ... $\longleftrightarrow u \in [up\ M]_{v,n}$
unfolding *DBM-zone-repr-def* **by** *blast*
finally show *?thesis* **.**

next
  case *False*
  then have *M 0 0 < 0*
    by *auto*
  then have *up M 0 0 < 0*
    unfolding *up-def* **by** *auto*
  with ‹*M 0 0 < 0*› **have** $[M]_{v,n} = \{\}$ $[up\ M]_{v,n} = \{\}$
    by (*auto intro*!: *DBM-triv-emptyI*)
  then show *?thesis*
    unfolding *zone-delay-def* **by** *blast*
qed

end

5 Implementation of DBM Operations

theory *DBM-Operations-Impl*
  imports
    DBM-Operations
    DBM-Normalization
    Refine-Imperative-HOL.IICF
    HOL−Library.IArray
begin

5.1 Misc

lemma *fold-last*:
  fold f (*xs* @ [*x*]) *a* = *f x* (*fold f xs a*)
by *simp*

5.2 Reset

definition
  reset-canonical M 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 Le d* + *M 0 j*
        else if i ≠ *k* ∧ *j* = *k then Le* (−*d*) + *M i 0*
        else M i j
)

— However, DBM entries are NOT a member of this typeclass.
lemma *canonical-is-cyc-free*:
 fixes *M* :: *nat* ⇒ *nat* ⇒ (′*b* :: {*linordered-cancel-ab-semigroup-add*, *linordered-ab-monoid-add*})

assumes *canonical M n*
shows *cyc-free M n*
proof (*cases* ∀ *i* ≤ *n*. *0* ≤ *M i i*)
case *True*
with *assms* **show** *?thesis* **by** (*rule canonical-cyc-free*)
next
case *False*
then obtain *i* **where** *i* ≤ *n M i i* < *0* **by** *auto*
then have *M i i* + *M i i* < *M i i* **using** *add-strict-left-mono* **by** *fastforce*
with ‹*i* ≤ *n*› *assms* **show** *?thesis* **by** *fastforce*
qed

lemma *dbm-neg-add*:
fixes *a* :: (′*t* :: *time*) *DBMEntry*
assumes *a* < *0*
shows *a* + *a* < *0*
using *assms* **unfolding** *neutral add less*
by (*cases a*) *auto*

instance *linordered-ab-group-add* ⊆ *linordered-cancel-ab-monoid-add* **by** *standard auto*

lemma *Le-cancel-1*[*simp*]:
fixes *d* :: ′*c* :: *linordered-ab-group-add*
shows *Le d* + *Le* (−*d*) = *Le 0*
unfolding *add* **by** *simp*

lemma *Le-cancel-2*[*simp*]:
fixes *d* :: ′*c* :: *linordered-ab-group-add*
shows *Le* (−*d*) + *Le d* = *Le 0*
unfolding *add* **by** *simp*

lemma *reset-canonical-canonical*′:
canonical (*reset-canonical M k* (*d* :: ′*c* :: *linordered-ab-group-add*)) *n*
if *M 0 0* = *0 M k k* = *0 canonical M n k* > *0* **for** *k n* :: *nat*
proof −
have *add-mono-neutr*′: *a* ≤ *a* + *b* **if** *b* ≥ *Le* (*0* :: ′*c*) **for** *a b*
using *that* **unfolding** *neutral*[*symmetric*] **by** (*simp add*: *add-increasing2*)
have *add-mono-neutl*′: *a* ≤ *b* + *a* **if** *b* ≥ *Le* (*0* :: ′*c*) **for** *a b*
using *that* **unfolding** *neutral*[*symmetric*] **by** (*simp add*: *add-increasing*)
show *?thesis*
using *that*
unfolding *reset-canonical-def neutral*
apply (*clarsimp split*: *if-splits*)

```
apply safe
                  apply (simp add: add-mono-neutr'; fail)
                 apply (simp add: comm; fail)
                apply (simp add: add-mono-neutl'; fail)
               apply (simp add: comm; fail)
              apply (simp add: add-mono-neutl'; fail)
             apply (simp add: add-mono-neutl'; fail)
            apply (simp add: add-mono-neutl'; fail)
           apply (simp add: add-mono-neutl' add-mono-neutr'; fail)
      apply (simp add: add.assoc[symmetric] add-mono-neutl' add-mono-neutr';
fail)
     apply (simp add: add.assoc[symmetric] add-mono-neutl' add-mono-neutr'
comm; fail)
    apply (simp add: add.assoc[symmetric] add-mono-neutl' add-mono-neutr';
fail)
subgoal premises prems for i j k
proof -
  from prems have M i k ≤ M i 0 + M 0 k
    by auto
  also have ... ≤ Le (- d) + M i 0 + (Le d + M 0 k)
   apply (simp add: add.assoc[symmetric], simp add: comm, simp add:
add.assoc[symmetric])
    using prems(1) that(1) by auto
  finally show ?thesis .
qed
subgoal premises prems for i j k
proof -
  from prems have Le 0 ≤ M 0 j + M j 0
    by force
  also have ... ≤ Le d + M 0 j + (Le (- d) + M j 0)
   apply (simp add: add.assoc[symmetric], simp add: comm, simp add:
add.assoc[symmetric])
    using prems(1) that(1) by (auto simp: add.commute)
  finally show ?thesis .
qed
subgoal premises prems for i j k
proof -
  from prems have Le 0 ≤ M 0 j + M j 0
    by force
  then show ?thesis
    by (simp add: add.assoc add-mono-neutr')
qed
subgoal premises prems for i j k
proof -
```

```
      from prems have M 0 k ≤ M 0 j + M j k
        by force
      then show ?thesis
        by (simp add: add-left-mono add.assoc)
    qed
    subgoal premises prems for i j
    proof −
      from prems have M i 0 ≤ M i j + M j 0
        by force
      then show ?thesis
      by (simp add: ab-semigroup-add-class.add.left-commute add-mono-right)
    qed
    subgoal premises prems for i j
    proof −
      from prems have Le 0 ≤ M 0 j + M j 0
        by force
      then show ?thesis
      by (simp add: ab-semigroup-add-class.add.left-commute add-mono-neutr')
    qed
    subgoal premises prems for i j
    proof −
      from prems have M i 0 ≤ M i j + M j 0
        by force
      then show ?thesis
      by (simp add: ab-semigroup-add-class.add.left-commute add-mono-right)
    qed
    done
qed

lemma reset-canonical-canonical:
  canonical (reset-canonical M k (d :: 'c :: linordered-ab-group-add)) n
  if ∀ i ≤ n. M i i = 0 canonical M n k > 0 for k n :: nat
proof −
  have add-mono-neutr': a ≤ a + b if b ≥ Le (0 :: 'c) for a b
   using that unfolding neutral[symmetric] by (simp add: add-increasing2)
  have add-mono-neutl': a ≤ b + a if b ≥ Le (0 :: 'c) for a b
   using that unfolding neutral[symmetric] by (simp add: add-increasing)
  show ?thesis
    using that
    unfolding reset-canonical-def neutral
    apply (clarsimp split: if-splits)
    apply safe
                 apply (simp add: add-mono-neutr'; fail)
                apply (simp add: comm; fail)
```

apply (*simp add*: *add-mono-neutl′*; *fail*)
apply (*simp add*: *comm*; *fail*)
apply (*simp add*: *add-mono-neutl′*; *fail*)
apply (*simp add*: *add-mono-neutl′*; *fail*)
apply (*simp add*: *add-mono-neutl′*; *fail*)
apply (*simp add*: *add-mono-neutl′ add-mono-neutr′*; *fail*)
apply (*simp add*: *add.assoc*[*symmetric*] *add-mono-neutl′ add-mono-neutr′*; *fail*)
apply (*simp add*: *add.assoc*[*symmetric*] *add-mono-neutl′ add-mono-neutr′ comm*; *fail*)
apply (*simp add*: *add.assoc*[*symmetric*] *add-mono-neutl′ add-mono-neutr′*; *fail*)
subgoal premises *prems* **for** *i j k*
proof −
from *prems* **have** $M\ i\ k \leq M\ i\ 0 + M\ 0\ k$
by *auto*
also have $\ldots \leq Le\ (-\ d) + M\ i\ 0 + (Le\ d + M\ 0\ k)$
apply (*simp add*: *add.assoc*[*symmetric*], *simp add*: *comm*, *simp add*: *add.assoc*[*symmetric*])
using *prems*(*1*) *that*(*1*) **by** (*auto simp*: *add.commute*)
finally show *?thesis* **.**
qed
subgoal premises *prems* **for** *i j k*
proof −
from *prems* **have** $Le\ 0 \leq M\ 0\ j + M\ j\ 0$
by *force*
also have $\ldots \leq Le\ d + M\ 0\ j + (Le\ (-\ d) + M\ j\ 0)$
apply (*simp add*: *add.assoc*[*symmetric*], *simp add*: *comm*, *simp add*: *add.assoc*[*symmetric*])
using *prems*(*1*) *that*(*1*) **by** (*auto simp*: *add.commute*)
finally show *?thesis* **.**
qed
subgoal premises *prems* **for** *i j k*
proof −
from *prems* **have** $Le\ 0 \leq M\ 0\ j + M\ j\ 0$
by *force*
then show *?thesis*
by (*simp add*: *add.assoc add-mono-neutr′*)
qed
subgoal premises *prems* **for** *i j k*
proof −
from *prems* **have** $M\ 0\ k \leq M\ 0\ j + M\ j\ k$
by *force*
then show *?thesis*

```
      by (simp add: add-left-mono add.assoc)
    qed
    subgoal premises prems for i j
    proof -
      from prems have M i 0 ≤ M i j + M j 0
        by force
      then show ?thesis
      by (simp add: ab-semigroup-add-class.add.left-commute add-mono-right)
    qed
    subgoal premises prems for i j
    proof -
      from prems have Le 0 ≤ M 0 j + M j 0
        by force
      then show ?thesis
      by (simp add: ab-semigroup-add-class.add.left-commute add-mono-neutr')
    qed
    subgoal premises prems for i j
    proof -
      from prems have M i 0 ≤ M i j + M j 0
        by force
      then show ?thesis
      by (simp add: ab-semigroup-add-class.add.left-commute add-mono-right)
    qed
    done
qed


lemma canonicalD[simp]:
  assumes canonical M n i ≤ n j ≤ n k ≤ n
  shows min (dbm-add (M i k) (M k j)) (M i j) = M i j
using assms unfolding add[symmetric] min-def by fastforce

lemma reset-reset-canonical:
  assumes canonical M n k > 0 k ≤ n clock-numbering v
  shows [reset M n k d]_{v,n} = [reset-canonical M k d]_{v,n}
proof safe
  fix u assume u ∈ [reset M n k d]_{v,n}
  show u ∈ [reset-canonical M k d]_{v,n}
  unfolding DBM-zone-repr-def DBM-val-bounded-def
  proof (safe, goal-cases)
    case 1
    with ‹u ∈ -› have
      Le 0 ≤ reset M n k d 0 0
    unfolding DBM-zone-repr-def DBM-val-bounded-def less-eq by auto
```

also have . . . = *M 0 0* **unfolding** *reset-def* **using** *assms* **by** *auto*
finally show *?case* **unfolding** *less-eq reset-canonical-def* **using** ‹*k* > *0*› **by** *auto*
next
case (*2 c*)
from ‹*clock-numbering -*› **have** *v c* > *0* **by** *auto*
show *?case*
proof (*cases v c* = *k*)
case *True*
with ‹*v c* > *0*› ‹*u* ∈ -› ‹*v c* ≤ *n*› **show** *?thesis*
unfolding *reset-canonical-def DBM-zone-repr-def DBM-val-bounded-def reset-def* **by** *auto*
next
case *False*
show *?thesis*
proof (*cases v c* = *k*)
case *True*
with ‹*v c* > *0*› ‹*u* ∈ -› ‹*v c* ≤ *n*› ‹*k* > *0*› **show** *?thesis*
unfolding *reset-canonical-def DBM-zone-repr-def DBM-val-bounded-def reset-def*
by *auto*
next
case *False*
with ‹*v c* > *0*› ‹*k* > *0*› ‹*v c* ≤ *n*› ‹*k* ≤ *n*› ‹*canonical - -*› ‹*u* ∈ -› **have**
dbm-entry-val u None (*Some c*) (*M 0* (*v c*))
unfolding *DBM-zone-repr-def DBM-val-bounded-def reset-def* **by** *auto*
with *False* ‹*k* > *0*› **show** *?thesis* **unfolding** *reset-canonical-def* **by** *auto*
qed
qed
next
case (*3 c*)
from ‹*clock-numbering -*› **have** *v c* > *0* **by** *auto*
show *?case*
proof (*cases v c* = *k*)
case *True*
with ‹*v c* > *0*› ‹*u* ∈ -› ‹*v c* ≤ *n*› **show** *?thesis*
unfolding *reset-canonical-def DBM-zone-repr-def DBM-val-bounded-def reset-def* **by** *auto*
next
case *False*
show *?thesis*

proof (*cases v c = k*)
case *True*
with ‹*v c > 0*› ‹*u ∈ -*› ‹*v c ≤ n*› ‹*k > 0*› **show** *?thesis*
unfolding *reset-canonical-def DBM-zone-repr-def DBM-val-bounded-def reset-def*
by *auto*
next
case *False*
with ‹*v c > 0*› ‹*k > 0*› ‹*v c ≤ n*› ‹*k ≤ n*› ‹*canonical - -*› ‹*u ∈ -*› **have**
dbm-entry-val u (Some c) None (M (v c) 0)
unfolding *DBM-zone-repr-def DBM-val-bounded-def reset-def* **by** *auto*
with *False* ‹*k > 0*› **show** *?thesis* **unfolding** *reset-canonical-def* **by** *auto*
qed
qed
next
case (*4 c1 c2*)
from ‹*clock-numbering -*› **have** *v c1 > 0 v c2 > 0* **by** *auto*
show *?case*
proof (*cases v c1 = k*)
case *True*
show *?thesis*
proof (*cases v c2 = k*)
case *True*
with ‹*v c1 = k*› ‹*v c1 > 0*› ‹*v c2 > 0*› ‹*u ∈ -*› ‹*v c1 ≤ n*› ‹*v c2 ≤ n*› ‹*canonical - -*›
have *reset-canonical M k d (v c1) (v c2) = M k k*
unfolding *reset-canonical-def* **by** *auto*
moreover from *True* ‹*v c1 = k*› ‹*v c1 > 0*› ‹*v c2 > 0*› ‹*v c1 ≤ n*› ‹*v c2 ≤ n*›
have *reset M n k d (v c1) (v c2) = M k k* **unfolding** *reset-def* **by** *auto*
moreover from ‹*u ∈ -*› ‹*v c1 = k*› ‹*v c2 = k*› ‹*k ≤ n*› **have**
dbm-entry-val u (Some c1) (Some c2) (reset M n k d k k)
unfolding *DBM-zone-repr-def DBM-val-bounded-def* **by** *auto metis*
ultimately show *?thesis* **using** ‹*v c1 = k*› ‹*v c2 = k*› **by** *auto*
next
case *False*
with ‹*v c1 = k*› ‹*v c1 > 0*› ‹*k > 0*› ‹*v c1 ≤ n*› ‹*k ≤ n*› ‹*canonical - -*› ‹*u ∈ -*› **have**
dbm-entry-val u (Some c1) None (Le d)
unfolding *DBM-zone-repr-def DBM-val-bounded-def reset-def* **by**

auto
moreover from ‹*v c2* ≠ *k*› ‹*k* > *0*› ‹*v c2* ≤ *n*› ‹*k* ≤ *n*› ‹*canonical* - -› ‹*u* ∈ -› **have**
dbm-entry-val u None (*Some c2*) (*M 0* (*v c2*))
unfolding *DBM-zone-repr-def DBM-val-bounded-def reset-def* **by** *auto*
ultimately show *?thesis* **using** *False* ‹*k* > *0*› ‹*v c1* = *k*› ‹*v c2* > *0*›
unfolding *reset-canonical-def add* **by** (*auto intro*: *dbm-entry-val-add-4*)
qed
next
case *False*
show *?thesis*
proof (*cases v c2* = *k*)
case *True*
from ‹*v c1* ≠ *k*› ‹*v c1* > *0*› ‹*k* > *0*› ‹*v c1* ≤ *n*› ‹*k* ≤ *n*› ‹*canonical* - -› ‹*u* ∈ -› **have**
dbm-entry-val u (*Some c1*) *None* (*M* (*v c1*) *0*)
unfolding *DBM-zone-repr-def DBM-val-bounded-def reset-def* **by** *auto*
moreover from ‹*v c2* = *k*› ‹*k* > *0*› ‹*v c2* ≤ *n*› ‹*k* ≤ *n*› ‹*canonical* - -› ‹*u* ∈ -› **have**
dbm-entry-val u None (*Some c2*) (*Le* (−*d*))
unfolding *DBM-zone-repr-def DBM-val-bounded-def reset-def* **by** *auto*
ultimately show *?thesis* **using** *False* ‹*k* > *0*› ‹*v c2* = *k*› ‹*v c1* > *0*› ‹*v c2* > *0*›
unfolding *reset-canonical-def*
apply *simp*
apply (*subst add.commute*)
by (*auto intro*: *dbm-entry-val-add-4*[*folded add*])
next
case *False*
from ‹*u* ∈ -› ‹*v c1* ≤ *n*› ‹*v c2* ≤ *n*› **have**
dbm-entry-val u (*Some c1*) (*Some c2*) (*reset M n k d* (*v c1*) (*v c2*))
unfolding *DBM-zone-repr-def DBM-val-bounded-def* **by** *auto*
with ‹*v c1* ≠ *k*› ‹*v c2* ≠ *k*› ‹*v c1* ≤ *n*› ‹*v c2* ≤ *n*› ‹*k* ≤ *n*› ‹*canonical* - -› **have**
dbm-entry-val u (*Some c1*) (*Some c2*) (*M* (*v c1*) (*v c2*))
unfolding *DBM-zone-repr-def DBM-val-bounded-def reset-def* **by** *auto*
with ‹*v c1* ≠ *k*› ‹*v c2* ≠ *k*› **show** *?thesis* **unfolding** *reset-canonical-def* **by** *auto*
qed

qed
qed
next
fix *u* **assume** $u \in [reset\text{-}canonical\ M\ k\ d]_{v,n}$
note *unfolds* = *DBM-zone-repr-def DBM-val-bounded-def reset-canonical-def*
show $u \in [reset\ M\ n\ k\ d]_{v,n}$
unfolding *DBM-zone-repr-def DBM-val-bounded-def*
proof (*safe*, *goal-cases*)
case *1*
with ‹*u* ∈ -› **have**
Le 0 ≤ *reset-canonical M k d 0 0*
unfolding *DBM-zone-repr-def DBM-val-bounded-def less-eq* **by** *auto*
also have . . . = *M 0 0* **unfolding** *reset-canonical-def* **using** *assms* **by** *auto*
finally show *?case* **unfolding** *less-eq reset-def* **using** ‹*k* > *0*› ‹*k* ≤ *n*› ‹*canonical - -*› **by** *auto*
next
case (*2 c*)
with *assms* **have** *v c* > *0* **by** *auto*
note *A* = *this assms*(*1*−*3*) ‹*v c* ≤ *n*›
show *?case*
proof (*cases v c* = *k*)
case *True*
with *A* ‹*u* ∈ -› **show** *?thesis* **unfolding** *reset-def unfolds* **by** *auto*
next
case *False*
with *A* ‹*u* ∈ -› **show** *?thesis* **unfolding** *unfolds reset-def* **by** *auto*
qed
next
case (*3 c*)
with *assms* **have** *v c* > *0* **by** *auto*
note *A* = *this assms*(*1*−*3*) ‹*v c* ≤ *n*›
show *?case*
proof (*cases v c* = *k*)
case *True*
with *A* ‹*u* ∈ -› **show** *?thesis* **unfolding** *reset-def unfolds* **by** *auto*
next
case *False*
with *A* ‹*u* ∈ -› **show** *?thesis* **unfolding** *unfolds reset-def* **by** *auto*
qed
next
case (*4 c1 c2*)
with *assms* **have** *v c1* > *0 v c2* > *0* **by** *auto*
note *A* = *this assms*(*1*−*3*) ‹*v c1* ≤ *n*› ‹*v c2* ≤ *n*›

```
    show ?case
    proof (cases v c1 = k)
      case True
      show ?thesis
      proof (cases v c2 = k)
        case True
        with ‹u ∈ -› A ‹v c1 = k› have
          dbm-entry-val u (Some c1) (Some c2) (reset-canonical M k d k k)
        unfolding DBM-zone-repr-def DBM-val-bounded-def by auto metis
        with A ‹v c1 = k› have
          dbm-entry-val u (Some c1) (Some c2) (M k k)
        unfolding reset-canonical-def by auto
       with A ‹v c1 = k› show ?thesis unfolding reset-def unfolds by auto
      next
        case False
       with A ‹v c1 = k› show ?thesis unfolding reset-def unfolds by auto
      qed
    next
      case False
      show ?thesis
      proof (cases v c2 = k)
        case False
        with ‹u ∈ -› A ‹v c1 ≠ k› have
           dbm-entry-val u (Some c1) (Some c2) (reset-canonical M k d (v
c1) (v c2))
        unfolding DBM-zone-repr-def DBM-val-bounded-def by auto
        with A ‹v c1 ≠ k› ‹v c2 ≠ k› have
          dbm-entry-val u (Some c1) (Some c2) (M (v c1) (v c2))
        unfolding reset-canonical-def by auto
       with A ‹v c1 ≠ k› show ?thesis unfolding reset-def unfolds by auto
      next
        case True
       with A ‹v c1 ≠ k› show ?thesis unfolding reset-def unfolds by auto
      qed
    qed
  qed
qed

lemma reset-canonical-diag-preservation:
  fixes k :: nat
  assumes k > 0
  shows ∀ i ≤ n. (reset-canonical M k d) i i = M i i
using assms unfolding reset-canonical-def by auto
```

definition *reset''* **where**
reset'' M n cs v d = *fold* (λ *c M*. *reset-canonical M* (*v c*) *d*) *cs M*

lemma *reset''-diag-preservation*:
assumes *clock-numbering v*
shows $\forall$ *i* $\leq$ *n*. (*reset'' M n cs v d*) *i i* = *M i i*
using *assms*
apply (*induction cs arbitrary*: *M*)
unfolding *reset''-def* **apply** *auto*[]
using *reset-canonical-diag-preservation* **by** *simp blast*

lemma *reset-resets*:
assumes $\forall$ *k*$\leq$*n*. *k* > *0* $\longrightarrow$ ($\exists$ *c*. *v c* = *k*) *clock-numbering' v n v c* $\leq$ *n*
shows [*reset M n* (*v c*) *d*]$_{v,n}$ = {*u*(*c* := *d*) | *u*. *u* $\in$ [*M*]$_{v,n}$}
proof *safe*
fix *u* **assume** *u*: *u* $\in$ [*reset M n* (*v c*) *d*]$_{v,n}$
with *assms* **have**
u c = *d*
by (*auto intro*: *DBM-reset-sound2*[*OF - DBM-reset-reset*] *simp*: *DBM-zone-repr-def*)
moreover from *DBM-reset-sound*[*OF assms u*] **obtain** *d'* **where**
u(*c* := *d'*) $\in$ [*M*]$_{v,n}$ (**is** *?u* $\in$ -)
by *auto*
ultimately have *u* = *?u*(*c* := *d*) **by** *auto*
with ‹*?u* $\in$ -› **show** $\exists$ *u'*. *u* = *u'*(*c* := *d*) $\wedge$ *u'* $\in$ [*M*]$_{v,n}$ **by** *blast*
next
fix *u* **assume** *u*: *u* $\in$ [*M*]$_{v,n}$
with *DBM-reset-complete*[*OF assms*(*2*,*3*) *DBM-reset-reset*] *assms*
show *u*(*c* := *d*) $\in$ [*reset M n* (*v c*) *d*]$_{v,n}$ **unfolding** *DBM-zone-repr-def*
by *auto*
qed

lemma *reset-eq'*:
assumes *prems*: $\forall$ *k*$\leq$*n*. *k* > *0* $\longrightarrow$ ($\exists$ *c*. *v c* = *k*) *clock-numbering' v n v*
c $\leq$ *n*
and *eq*: [*M*]$_{v,n}$ = [*M'*]$_{v,n}$
shows [*reset M n* (*v c*) *d*]$_{v,n}$ = [*reset M' n* (*v c*) *d*]$_{v,n}$
using *reset-resets*[*OF prems*] *eq* **by** *blast*

lemma *reset-eq*:
assumes *prems*: $\forall$ *k*$\leq$*n*. *k* > *0* $\longrightarrow$ ($\exists$ *c*. *v c* = *k*) *clock-numbering' v n*
and *k*: *k* > *0 k* $\leq$ *n*
and *eq*: [*M*]$_{v,n}$ = [*M'*]$_{v,n}$
shows [*reset M n k d*]$_{v,n}$ = [*reset M' n k d*]$_{v,n}$
using *reset-eq'*[*OF prems - eq*] *prems*(*1*) *k* **by** *blast*

lemma *FW-reset-commute*:
  assumes *prems*: $\forall k \le n.\ k > 0 \longrightarrow (\exists c.\ v\ c = k)$ *clock-numbering'* $v\ n\ k > 0\ k \le n$
  shows $[FW\ (reset\ M\ n\ k\ d)\ n]_{v,n} = [reset\ (FW\ M\ n)\ n\ k\ d]_{v,n}$
using *reset-eq*[*OF prems*] *FW-zone-equiv*[*OF prems*(*1*)] **by** *blast*

lemma *reset-canonical-diag-presv*:
  fixes $k :: nat$
  assumes $M\ i\ i = Le\ 0\ k > 0$
  shows $(reset\text{-}canonical\ M\ k\ d)\ i\ i = Le\ 0$
unfolding *reset-canonical-def* **using** *assms* **by** *auto*

lemma *reset-cycle-free*:
  fixes $M :: ('t :: time)\ DBM$
  assumes *cycle-free M n*
    and *prems*: $\forall k \le n.\ k > 0 \longrightarrow (\exists c.\ v\ c = k)$ *clock-numbering'* $v\ n\ k > 0\ k \le n$
  shows *cycle-free* (*reset M n k d*) *n*
proof –
  from *assms cyc-free-not-empty cycle-free-diag-equiv* **have** $[M]_{v,n} \ne \{\}$ **by** *metis*
  with *reset-resets*[*OF prems*(*1*,*2*)] *prems*(*1*,*3*,*4*) **have** $[reset\ M\ n\ k\ d]_{v,n} \ne \{\}$ **by** *fast*
  with *not-empty-cyc-free*[*OF prems*(*1*)] *cycle-free-diag-equiv* **show** *?thesis* **by** *metis*
qed

lemma *reset'-reset''-equiv*:
  assumes *canonical M n* $d \ge 0\ \forall i \le n.\ M\ i\ i = 0$
      clock-numbering' $v\ n\ \forall\ c \in set\ cs.\ v\ c \le n$
    and *surj*: $\forall\ k \le n.\ k > 0 \longrightarrow (\exists\ c.\ v\ c = k)$
  shows $[reset'\ M\ n\ cs\ v\ d]_{v,n} = [reset''\ M\ n\ cs\ v\ d]_{v,n}$
proof –
  from *assms*(*3*,*4*,*5*) *surj* **have**
    $\forall i \le n.\ M\ i\ i \ge 0\ M\ 0\ 0 = Le\ 0\ \forall\ c \in set\ cs.\ M\ (v\ c)\ (v\ c) = Le\ 0$
  unfolding *neutral* **by** *auto*
  note *assms* = *assms*(*1*,*2*) *this assms*(*4*–)
  from ‹*clock-numbering' v n*› **have** *clock-numbering v* **by** *auto*
  from *canonical-cyc-free assms*(*1*,*3*) *cycle-free-diag-equiv* **have** *cycle-free M n* **by** *metis*
  have $reset'\ M\ n\ cs\ v\ d = foldr\ (\lambda\ c\ M.\ reset\ M\ n\ (v\ c)\ d)\ cs\ M$ **by** (*induction cs*) *auto*
  then have

$[FW\ (reset'\ M\ n\ cs\ v\ d)\ n]_{v,n} = [FW\ (foldr\ (\lambda\ c\ M.\ reset\ M\ n\ (v\ c)\ d)\ cs\ M)\ n]_{v,n}$
by *simp*
also have ... $= [foldr\ (\lambda c\ M.\ reset\text{-}canonical\ M\ (v\ c)\ d)\ cs\ M]_{v,n}$
using *assms*
apply (*induction cs*)
apply (*force simp*: *FW-canonical-id*)
apply *simp*
subgoal premises *prems* **for** *a cs*
proof −
let *?l* = *FW* (*reset* (*foldr* (λ*c M*. *reset M n* (*v c*) *d*) *cs M*) *n* (*v a*) *d*) *n*
let *?m* = *reset* (*foldr* (λ*c M*. *reset-canonical M* (*v c*) *d*) *cs M*) *n* (*v a*) *d*
let *?r* = *reset-canonical* (*foldr* (λ*c M*. *reset-canonical M* (*v c*) *d*) *cs M*) (*v a*) *d*
have *foldr* (λ*c M*. *reset-canonical M* (*v c*) *d*) *cs M 0 0* = *Le 0*
apply (*induction cs*)
using *prems* **by** (*force intro*: *reset-canonical-diag-presv*)+
from *prems*(*6*) **have** *canonical* (*foldr* (λ*c M*. *reset-canonical M* (*v c*) *d*) *cs M*) *n*
apply (*induction cs*)
using ‹*canonical M n*› **apply** *force*
apply *simp*
apply (*rule reset-canonical-canonical'*[*unfolded neutral*])
using *assms* **apply** *simp*
subgoal premises − **for** *a cs*
apply (*induction cs*)
using *assms*(*4*) ‹*clock-numbering v*› **by** (*force intro*: *reset-canonical-diag-presv*)+
subgoal premises *prems* **for** *a cs*
apply (*induction cs*)
using *prems* ‹*clock-numbering v*› **by** (*force intro*: *reset-canonical-diag-presv*)+
apply (*simp*; *fail*)
using ‹*clock-numbering v*› **by** *metis*
have $[FW\ (reset\ (foldr\ (\lambda c\ M.\ reset\ M\ n\ (v\ c)\ d)\ cs\ M)\ n\ (v\ a)\ d)\ n]_{v,n}$
$= [reset\ (FW\ (foldr\ (\lambda c\ M.\ reset\ M\ n\ (v\ c)\ d)\ cs\ M)\ n)\ n\ (v\ a)\ d]_{v,n}$
using *assms*(*8*−) *prems*(*7*−) **by** − (*rule FW-reset-commute*; *auto*)
also from *prems* **have** ... $= [?m]_{v,n}$ **by** − (*rule reset-eq*; *auto*)
also from ‹*canonical* (*foldr* - - -) *n*› *prems* **have**
... $= [?r]_{v,n}$
by − (*rule reset-reset-canonical*; *simp*)
finally show *?thesis* **.**
qed

done
also have ... = $[reset'' \ M \ n \ cs \ v \ d]_{v,n}$ **unfolding** *reset''-def*
apply (*rule arg-cong*[**where** $f = \lambda \ M. \ [M]_{v,n}$])
apply (*rule fun-cong*[**where** $x = M$])
apply (*rule foldr-fold*)
apply (*rule ext*)
apply *simp*
subgoal for *x y M*
proof −
from ‹*clock-numbering v*› **have** *v x > 0 v y > 0* **by** *auto*
show *?thesis*
proof (*cases v x = v y*)
case *True*
then show *?thesis* **unfolding** *reset-canonical-def* **by** *force*
next
case *False*
with ‹*v x > 0*› ‹*v y > 0*› **show** *?thesis* **unfolding** *reset-canonical-def* **by** *fastforce*
qed
qed
done
finally show *?thesis* **using** *FW-zone-equiv*[*OF surj*] **by** *metis*
qed

Eliminating the clock numbering

definition *reset'''* **where**
reset''' M n cs d = fold (λ *c M. reset-canonical M c d*) *cs M*

lemma *reset''-reset'''*:
assumes ∀ *c* ∈ *set cs. v c = c*
shows *reset'' M n cs v d = reset''' M n cs d*
using *assms*
apply (*induction cs arbitrary*: *M*)
unfolding *reset''-def reset'''-def* **by** *simp+*

type-synonym *'a DBM' = nat × nat ⇒ 'a DBMEntry*

definition
reset-canonical-upd
(*M* :: (*'a* :: {*linordered-cancel-ab-monoid-add*,*uminus*}) *DBM'*) (*n*:: *nat*) (*k*:: *nat*) *d* =
fold (λ *i M. if i = k then M else M*((*k*, *i*) := *Le d* + *M*(*0*,*i*), (*i*, *k*) := *Le* (−*d*) + *M*(*i*, *0*)))
(*map nat* [*1*..*n*])

($M((k,\ 0) := Le\ d,\ (0,\ k) := Le\ (-d))$)

lemma *one-upto-Suc*:
  $[1..<Suc\ i\ +\ 1] = [1..<i+1]$ @ $[Suc\ i]$
  by *simp*

lemma *one-upto-Suc′*:
  $[1..Suc\ i] = [1..i]$ @ $[Suc\ i]$
  by (*simp add*: *upto-rec2*)

lemma *one-upto-Suc′′*:
  $[1..1\ +\ i] = [1..i]$ @ $[Suc\ i]$
  by (*simp add*: *upto-rec2*)

lemma *reset-canonical-upd-diag-id*:
  fixes *k n* :: *nat*
  assumes $k > 0$
  shows (*reset-canonical-upd M n k d*) $(k,\ k) = M\ (k,\ k)$
unfolding *reset-canonical-upd-def* **using** *assms* **by** (*induction n*) (*auto simp*: *upto-rec2*)

lemma *reset-canonical-upd-out-of-bounds-id1*:
  fixes *i j k n* :: *nat*
  assumes $i \neq k$ $i > n$
  shows (*reset-canonical-upd M n k d*) $(i,\ j) = M\ (i,\ j)$
using *assms* **by** (*induction n*) (*auto simp add*: *reset-canonical-upd-def upto-rec2*)

lemma *reset-canonical-upd-out-of-bounds-id2*:
  fixes *i j k n* :: *nat*
  assumes $j \neq k$ $j > n$
  shows (*reset-canonical-upd M n k d*) $(i,\ j) = M\ (i,\ j)$
using *assms* **by** (*induction n*) (*auto simp add*: *reset-canonical-upd-def upto-rec2*)

lemma *reset-canonical-upd-out-of-bounds1*:
  fixes *i j k n* :: *nat*
  assumes $k \leq n$ $i > n$
  shows (*reset-canonical-upd M n k d*) $(i,\ j) = M\ (i,\ j)$
using *assms reset-canonical-upd-out-of-bounds-id1* **by** (*metis not-le*)

lemma *reset-canonical-upd-out-of-bounds2*:
  fixes *i j k n* :: *nat*
  assumes $k \leq n$ $j > n$
  shows (*reset-canonical-upd M n k d*) $(i,\ j) = M\ (i,\ j)$

using *assms reset-canonical-upd-out-of-bounds-id2* **by** (*metis not-le*)

lemma *reset-canonical-upd-id1*:
  fixes *k n* :: *nat*
  assumes $k > 0$ $i > 0$ $i \leq n$ $i \neq k$
  shows (*reset-canonical-upd M n k d*) (*i*, *k*) = *Le* (−*d*) + *M*(*i*,*0*)
using *assms* **apply** (*induction n*)
apply (*simp add*: *reset-canonical-upd-def*; *fail*)
subgoal for *n*
  apply (*simp add*: *reset-canonical-upd-def*)
  apply (*subst one-upto-Suc''*)
 using *reset-canonical-upd-out-of-bounds-id1*[*unfolded reset-canonical-upd-def*,
where *j* = *0* **and** *M* = *M*]
by *fastforce*
done

lemma *reset-canonical-upd-id2*:
  fixes *k n* :: *nat*
  assumes $k > 0$ $i > 0$ $i \leq n$ $i \neq k$
  shows (*reset-canonical-upd M n k d*) (*k*, *i*) = *Le d* + *M*(*0*,*i*)
unfolding *reset-canonical-upd-def* **using** *assms* **apply** (*induction n*)
apply (*simp add*: *upto-rec2*; *fail*)
subgoal for *n*
  apply (*simp add*: *one-upto-Suc''*)
 using *reset-canonical-upd-out-of-bounds-id2*[*unfolded reset-canonical-upd-def*,
where *i* = *0* **and** *M* = *M*]
by *fastforce*
done

lemma *reset-canonical-updid-0-1*:
  fixes *n* :: *nat*
  assumes $k > 0$
  shows (*reset-canonical-upd M n k d*) (*0*, *k*) = *Le* (−*d*)
using *assms* **by** (*induction n*) (*auto simp add*: *reset-canonical-upd-def upto-rec2*)

lemma *reset-canonical-updid-0-2*:
  fixes *n* :: *nat*
  assumes $k > 0$
  shows (*reset-canonical-upd M n k d*) (*k*, *0*) = *Le d*
using *assms* **by** (*induction n*) (*auto simp add*: *reset-canonical-upd-def upto-rec2*)

lemma *reset-canonical-upd-id*:
  fixes *n* :: *nat*
  assumes $i \neq k$ $j \neq k$

shows (*reset-canonical-upd M n k d*) (*i*,*j*) = *M* (*i*,*j*)
using *assms* **by** (*induction n*; *simp add*: *reset-canonical-upd-def upto-rec2*)

lemma *reset-canonical-upd-reset-canonical*:
fixes *i j k n* :: *nat* **and** *M* :: *nat* × *nat* ⇒ ('*a* :: {*linordered-cancel-ab-monoid-add*,*uminus*}) *DBMEntry*
assumes *k* > *0* *i* ≤ *n* *j* ≤ *n* ∀ *i* ≤ *n*. ∀ *j* ≤ *n*. *M* (*i*, *j*) = *M*′ *i j*
shows (*reset-canonical-upd M n k d*)(*i*,*j*) = (*reset-canonical M*′ *k d*) *i j*
(**is** *?M*(*i*,*j*) = -)
proof (*cases i* = *k*)
case *True*
show *?thesis*
proof (*cases j* = *k*)
case *True*
with ‹*i* = *k*› *assms reset-canonical-upd-diag-id*[**where** *M* = *M*] **show** *?thesis*
by (*auto simp*: *reset-canonical-def*)
next
case *False*
show *?thesis*
proof (*cases j* = *0*)
case *False*
with ‹*i* = *k*› ‹*j* ≠ *k*› *assms* **have**
?M (*i*,*j*) = *Le d* + *M*(*0*,*j*)
using *reset-canonical-upd-id2*[**where** *M* = *M*] **by** *fastforce*
with ‹*i* = *k*› ‹*j* ≠ *k*› ‹*j* ≠ *0*› *assms* **show** *?thesis* **unfolding** *reset-canonical-def* **by** *auto*
next
case *True*
with ‹*i* = *k*› ‹*k* > *0*› **show** *?thesis* **by** (*simp add*: *reset-canonical-updid-0-2* *reset-canonical-def*)
qed
qed
next
case *False*
show *?thesis*
proof (*cases j* = *k*)
case *True*
show *?thesis*
proof (*cases i* = *0*)
case *False*
with ‹*j* = *k*› ‹*i* ≠ *k*›*assms* **have**
?M (*i*,*j*) = *Le* (−*d*) + *M*(*i*,*0*)
using *reset-canonical-upd-id1*[**where** *M* = *M*] **by** *fastforce*

with ‹$j = k$› ‹$i \neq k$› ‹$i \neq 0$› *assms* **show** *?thesis* **unfolding** *reset-canonical-def* **by** *force*
next
case *True*
with ‹$j = k$› ‹$k > 0$› **show** *?thesis* **by** (*simp add*: *reset-canonical-updid-0-1 reset-canonical-def*)
qed
next
case *False*
with ‹$i \neq k$› *assms* **show** *?thesis* **by** (*simp add*: *reset-canonical-upd-id reset-canonical-def*)
qed
qed

lemma *reset-canonical-upd-reset-canonical'*:
fixes *i j k n* :: *nat*
assumes $k > 0$ $i \leq n$ $j \leq n$
shows (*reset-canonical-upd M n k d*)(*i*,*j*) = (*reset-canonical* (*curry M*) *k d*) *i j* (**is** *?M*(*i*,*j*) = -)
proof (*cases* $i = k$)
case *True*
show *?thesis*
proof (*cases* $j = k$)
case *True*
with ‹$i = k$› *assms reset-canonical-upd-diag-id* **show** *?thesis* **by** (*auto simp add*: *reset-canonical-def*)
next
case *False*
show *?thesis*
proof (*cases* $j = 0$)
case *False*
with ‹$i = k$› ‹$j \neq k$› *assms* **have**
?M (*i*,*j*) = *Le d* + *M*(*0*,*j*)
using *reset-canonical-upd-id2*[**where** *M* = *M*] **by** *fastforce*
with ‹$i = k$› ‹$j \neq k$› ‹$j \neq 0$› **show** *?thesis* **unfolding** *reset-canonical-def* **by** *simp*
next
case *True*
with ‹$i = k$› ‹$k > 0$› **show** *?thesis* **by** (*simp add*: *reset-canonical-updid-0-2 reset-canonical-def*)
qed
qed
next
case *False*

show *?thesis*
proof (*cases j = k*)
case *True*
show *?thesis*
proof (*cases i = 0*)
case *False*
with ‹*j = k*› ‹*i ≠ k*›*assms* **have**
?M (i,j) = Le (−d) + M(i,0)
using *reset-canonical-upd-id1*[**where** *M = M*] **by** *fastforce*
with ‹*j = k*› ‹*i ≠ k*› ‹*i ≠ 0*› **show** *?thesis* **unfolding** *reset-canonical-def*
by *simp*
next
case *True*
with ‹*j = k*› ‹*k > 0*› **show** *?thesis* **by** (*simp add*: *reset-canonical-updid-0-1*
reset-canonical-def)
qed
next
case *False*
with ‹*i ≠ k*› **show** *?thesis* **by** (*simp add*: *reset-canonical-upd-id re-*
set-canonical-def)
qed
qed

lemma *reset-canonical-upd-canonical*:
canonical (*curry* (*reset-canonical-upd M n k* (*d* :: ′*c* :: {*linordered-ab-group-add*,*uminus*})))
n
if ∀ *i ≤ n. M (i, i) = 0 canonical* (*curry M*) *n k > 0* **for** *k n* :: *nat*
using *reset-canonical-canonical*[*of n curry M k*] *that*
by (*auto simp*: *reset-canonical-upd-reset-canonical*′)

definition *reset*′*-upd* **where**
reset′*-upd M n cs d = fold* (*λ c M. reset-canonical-upd M n c d*) *cs M*

lemma *reset*′′′*-reset*′*-upd*:
fixes *n*:: *nat* **and** *cs* :: *nat list*
assumes ∀ *c ∈ set cs. c ≠ 0 i ≤ n j ≤ n* ∀ *i ≤ n.* ∀ *j ≤ n. M (i, j) =*
M′ *i j*
shows (*reset*′*-upd M n cs d*) (*i, j*) = (*reset*′′′ *M*′ *n cs d*) *i j*
using *assms*
apply (*induction cs arbitrary*: *M M*′)
unfolding *reset*′*-upd-def reset*′′′*-def*
apply (*simp*; *fail*)
subgoal for *c cs M M*′
using *reset-canonical-upd-reset-canonical*[**where** *M = M*] **by** *auto*

done

lemma *reset'''-reset'-upd'*:
fixes *n*:: *nat* **and** *cs* :: *nat list* **and** *M* :: (*'a* :: {*linordered-cancel-ab-monoid-add*,*uminus*}) *DBM'*
assumes ∀ *c* ∈ *set cs*. *c* ≠ *0* *i* ≤ *n* *j* ≤ *n*
shows (*reset'-upd M n cs d*) (*i*, *j*) = (*reset'''* (*curry M*) *n cs d*) *i j*
using *reset'''-reset'-upd*[**where** *M* = *M* **and** *M'* = *curry M*, *OF assms*]
by *simp*

lemma *reset'-upd-out-of-bounds1*:
fixes *i j k n* :: *nat*
assumes ∀ *c* ∈ *set cs*. *c* ≤ *n* *i* > *n*
shows (*reset'-upd M n cs d*) (*i*, *j*) = *M* (*i*, *j*)
using *assms*
by (*induction cs arbitrary*: *M*, *auto simp*: *reset'-upd-def intro*: *reset-canonical-upd-out-of-bounds-id1*)

lemma *reset'-upd-out-of-bounds2*:
fixes *i j k n* :: *nat*
assumes ∀ *c* ∈ *set cs*. *c* ≤ *n* *j* > *n*
shows (*reset'-upd M n cs d*) (*i*, *j*) = *M* (*i*, *j*)
using *assms*
by (*induction cs arbitrary*: *M*, *auto simp*: *reset'-upd-def intro*: *reset-canonical-upd-out-of-bounds-id2*)

lemma *reset-canonical-int-preservation*:
fixes *n* :: *nat*
assumes *dbm-int M n d* ∈ ℤ
shows *dbm-int* (*reset-canonical M k d*) *n*
using *assms* **unfolding** *reset-canonical-def* **by** (*auto dest*: *sum-not-inf-dest*)

lemma *reset-canonical-upd-int-preservation*:
assumes *dbm-int* (*curry M*) *n d* ∈ ℤ *k* > *0*
shows *dbm-int* (*curry* (*reset-canonical-upd M n k d*)) *n*
using *reset-canonical-int-preservation*[*OF assms*(*1*,*2*)] *reset-canonical-upd-reset-canonical'*
by (*metis assms*(*3*) *curry-conv*)

lemma *reset'-upd-int-preservation*:
assumes *dbm-int* (*curry M*) *n d* ∈ ℤ ∀ *c* ∈ *set cs*. *c* ≠ *0*
shows *dbm-int* (*curry* (*reset'-upd M n cs d*)) *n*
using *assms*
apply (*induction cs arbitrary*: *M*)
unfolding *reset'-upd-def*
apply (*simp*; *fail*)
apply (*drule reset-canonical-upd-int-preservation*; *auto*)

done

lemma *reset-canonical-upd-diag-preservation*:
  fixes *i* :: *nat*
  assumes $k > 0$
  shows $\forall\ i \le n.$ (*reset-canonical-upd M n k d*) (*i*, *i*) = *M* (*i*, *i*)
using *reset-canonical-diag-preservation reset-canonical-upd-reset-canonical′ assms*
by (*metis curry-conv*)

lemma *reset′-upd-diag-preservation*:
  assumes $\forall\ c \in$ *set cs*. $c > 0$ $i \le n$
  shows (*reset′-upd M n cs d*) (*i*, *i*) = *M* (*i*, *i*)
using *assms*
by (*induction cs arbitrary*: *M*; *simp add*: *reset′-upd-def reset-canonical-upd-diag-preservation*)

lemma *upto-from-1-upt*:
  fixes *n* :: *nat*
  shows *map nat* [*1..int n*] = [*1..<n+1*]
by (*induction n*) (*auto simp*: *one-upto-Suc″*)

lemma *reset-canonical-upd-alt-def*:
 reset-canonical-upd (*M* :: (*′a* :: {*linordered-cancel-ab-monoid-add*,*uminus*}) *DBM′*) (*n*:: *nat*) (*k* :: *nat*) *d* =
```
    fold
      (λ i M.
        if i = k then
          M
        else do {
          let m0i = op-mtx-get M(0,i);
          let mi0 = op-mtx-get M(i, 0);
          M((k, i) := Le d + m0i, (i, k) := Le (−d) + mi0)
        }
      )
      [1..<n+1]
      (M((k, 0) := Le d, (0, k) := Le (−d)))
```

unfolding *reset-canonical-upd-def* **by** (*simp add*: *upto-from-1-upt cong*: *if-cong*)

5.3 Relaxation

named-theorems *dbm-entry-simps*

lemma [*dbm-entry-simps*]:
  $a + \infty = \infty$
unfolding *add* **by** (*cases a*) *auto*

lemma [*dbm-entry-simps*]:
  $\infty + b = \infty$
unfolding *add* **by** (*cases b*) *auto*

lemmas *any-le-inf*[*dbm-entry-simps*]

lemma *up-canonical-preservation*:
  assumes *canonical M n*
  shows *canonical* (*up M*) *n*
unfolding *up-def* **using** *assms* **by** (*simp add*: *dbm-entry-simps*)

definition *up-canonical* :: *'t DBM* ⇒ *'t DBM* **where**
  up-canonical M = (λ *i j*. *if* $i > 0 \wedge j = 0$ *then* ∞ *else M i j*)

lemma *up-canonical-eq-up*:
  assumes *canonical M n* $i \leq n$ $j \leq n$
  shows *up-canonical M i j* = *up M i j*
unfolding *up-canonical-def up-def* **using** *assms* **by** *simp*

lemma *DBM-up-to-equiv*:
  assumes $\forall\ i \leq n.\ \forall\ j \leq n.\ M\ i\ j = M'\ i\ j$
  shows $[M]_{v,n} = [M']_{v,n}$
 apply *safe*
 apply (*rule DBM-le-subset*)
using *assms* **by** (*auto simp*: *add*[*symmetric*] *intro*: *DBM-le-subset*)

lemma *up-canonical-equiv-up*:
  assumes *canonical M n*
  shows $[up\text{-}canonical\ M]_{v,n} = [up\ M]_{v,n}$
 apply (*rule DBM-up-to-equiv*)
unfolding *up-canonical-def up-def* **using** *assms* **by** *simp*

lemma *up-canonical-diag-preservation*:
  assumes $\forall\ i \leq n.\ M\ i\ i = 0$
  shows $\forall\ i \leq n.$ (*up-canonical M*) $i\ i = 0$
unfolding *up-canonical-def* **using** *assms* **by** *auto*

no-notation *Ref.update* (‹- := -› *62*)

definition *up-canonical-upd* :: *'t DBM'* ⇒ *nat* ⇒ *'t DBM'* **where**

up-canonical-upd M n = fold (λ *i M. M*((*i*,*0*) := ∞)) [*1*..<*n*+*1*] *M*

lemma *up-canonical-upd-rec*:
  up-canonical-upd M (*Suc n*) = (*up-canonical-upd M n*) ((*Suc n*, *0*) := ∞)
unfolding *up-canonical-upd-def* **by** *auto*

lemma *up-canonical-out-of-bounds1*:
  fixes *i* :: *nat*
  assumes *i* > *n*
  shows *up-canonical-upd M n* (*i*, *j*) = *M*(*i*,*j*)
using *assms* **by** (*induction n*) (*auto simp*: *up-canonical-upd-def*)

lemma *up-canonical-out-of-bounds2*:
  fixes *j* :: *nat*
  assumes *j* > *0*
  shows *up-canonical-upd M n* (*i*, *j*) = *M*(*i*,*j*)
using *assms* **by** (*induction n*) (*auto simp*: *up-canonical-upd-def*)

lemma *up-canonical-upd-up-canonical*:
  assumes *i* ≤ *n j* ≤ *n* ∀ *i* ≤ *n*. ∀ *j* ≤ *n*. *M* (*i*, *j*) = *M′ i j*
  shows (*up-canonical-upd M n*) (*i*, *j*) = (*up-canonical M′*) *i j*
using *assms*
proof (*induction n*)
  case *0*
  then show *?case* **by** (*simp add*: *up-canonical-upd-def up-canonical-def*;
fail)
next
  case (*Suc n*)
  show *?case*
  proof (*cases j* = *Suc n*)
    case *True*
    with *Suc.prems* **show** *?thesis* **by** (*simp add*: *up-canonical-out-of-bounds2*
up-canonical-def)
  next
    case *False*
    show *?thesis*
    proof (*cases i* = *Suc n*)
      case *True*
      with *Suc.prems* ‹*j* ≠ -› **show** *?thesis*
    by (*simp add*: *up-canonical-out-of-bounds1 up-canonical-def up-canonical-upd-rec*)
    next
      case *False*
      with *Suc* ‹*j* ≠ -› **show** *?thesis* **by** (*auto simp*: *up-canonical-upd-rec*)
    qed

qed
qed

lemma *up-canonical-int-preservation*:
assumes *dbm-int M n*
shows *dbm-int* (*up-canonical M*) *n*
using *assms* **unfolding** *up-canonical-def* **by** *auto*

lemma *up-canonical-upd-int-preservation*:
assumes *dbm-int* (*curry M*) *n*
shows *dbm-int* (*curry* (*up-canonical-upd M n*)) *n*
using *up-canonical-int-preservation*[*OF assms*] *up-canonical-upd-up-canonical*[**where** *M' = curry M*]
by (*auto simp*: *curry-def*)

lemma *up-canonical-diag-preservation'*:
(*up-canonical M*) *i i* = *M i i*
unfolding *up-canonical-def* **by** *auto*

lemma *up-canonical-upd-diag-preservation*:
(*up-canonical-upd M n*) (*i*, *i*) = *M* (*i*, *i*)
unfolding *up-canonical-upd-def* **by** (*induction n*) *auto*

5.4 Intersection

definition
unbounded-dbm n = (λ (*i*, *j*). (*if i* = *j* ∨ *i* > *n* ∨ *j* > *n then Le 0 else* ∞))

definition *And-upd* :: *nat* ⇒ ('t::{*linorder*,*zero*}) *DBM'* ⇒ '*t DBM'* ⇒ '*t DBM'* **where**
And-upd n A B =
fold (λ*i M*.
fold (λ*j M*. *M*((*i*,*j*) := *min* (*A*(*i*,*j*)) (*B*(*i*,*j*)))) [*0*..<*n*+*1*] *M*)
[*0*..<*n*+*1*]
(*unbounded-dbm n*)

lemma *fold-loop-inv-rule*:
assumes *I 0 x*
assumes ⋀*i x*. *I i x* ⟹ *i* ≤ *n* ⟹ *I* (*Suc i*) (*f i x*)
assumes ⋀*x*. *I n x* ⟹ *Q x*
shows *Q* (*fold f* [*0*..<*n*] *x*)
proof −
from *assms*(*2*) **have** *I n* (*fold f* [*0*..<*n*] *x*)

proof (*induction n*)
case *0*
show *?case*
by *simp* (*rule assms*)
next
case (*Suc n*)
show *?case*
using *Suc* **by** *auto*
qed
then show *?thesis*
by (*rule assms*(*3*))
qed

lemma *And-upd-min*:
assumes $i \leq n$ $j \leq n$
shows *And-upd n A B* $(i, j) = min\ (A(i,j))\ (B(i,j))$
unfolding *And-upd-def*
apply (*rule fold-loop-inv-rule*[**where** $I = \lambda k\ M.\ \forall\, i<k.\ \forall\, j \leq n.\ M(i,j) = min\ (A(i,j))\ (B(i,j))$])
apply (*simp*; *fail*)
subgoal for *k x*
apply (*rule fold-loop-inv-rule*[**where** $I =$
$\lambda j'\ M.\ \forall\, i \leq k.$
if $i = k$ *then*
$(\forall\, j < j'.\ M(i,j) = min\ (A(i,j))\ (B(i,j)))$
else
$(\forall\, j \leq n.\ M(i,j) = min\ (A(i,j))\ (B(i,j)))$])
by (*simp-all*) (*metis Suc-eq-plus1 less-Suc-eq-le*)
using *assms* **by** *auto*

lemma *And-upd-And*:
assumes $i \leq n$ $j \leq n$
$\forall\ i \leq n.\ \forall\ j \leq n.\ A\ (i, j) = A'\ i\ j$ $\forall\ i \leq n.\ \forall\ j \leq n.\ B\ (i, j) = B'\ i\ j$
shows *And-upd n A B* $(i, j) = And\ A'\ B'\ i\ j$
using *assms* **by** (*auto simp*: *And-upd-min*)

5.5 Inclusion

definition *pointwise-cmp* **where**
pointwise-cmp P n M $M' = (\forall\ i \leq n.\ \forall\ j \leq n.\ P\ (M\ i\ j)\ (M'\ i\ j))$

lemma *subset-eq-pointwise-le*:
fixes *M* :: *real DBM*
assumes *canonical M n* $\forall\ i \leq n.\ M\ i\ i = 0$ $\forall\ i \leq n.\ M'\ i\ i = 0$

and *prems*: *clock-numbering′ v n* $\forall k \le n.\ 0 < k \longrightarrow (\exists c.\ v\ c = k)$
shows $[M]_{v,n} \subseteq [M']_{v,n} \longleftrightarrow$ *pointwise-cmp* $(\le)$ *n M M′*
unfolding *pointwise-cmp-def*
apply *safe*
subgoal for *i j*
apply (*cases i = j*)
using *assms* **apply** (*simp*; *fail*)
apply (*rule DBM-canonical-subset-le*)
using *assms*(*1*−*3*) *prems* **by** (*auto simp*: *cyc-free-not-empty*[*OF canonical-cyc-free*])
by (*auto simp*: *less-eq intro*: *DBM-le-subset*)

definition *check-diag* :: *nat* ⇒ (*′t* :: {*linorder*, *zero*}) *DBM′* ⇒ *bool* **where**
check-diag n M ≡ ∃ *i* ≤ *n*. *M* (*i*, *i*) < *Le 0*

lemma *check-diag-empty*:
fixes *n* :: *nat* **and** *v*
assumes *surj*: $\forall\ k \le n.\ 0 < k \longrightarrow (\exists c.\ v\ c = k)$
assumes *check-diag n M*
shows $[curry\ M]_{v,n} = \{\}$
using *assms neg-diag-empty*[*OF surj*, **where** *M* = *curry M*] **unfolding** *check-diag-def neutral* **by** *auto*

lemma *check-diag-alt-def*:
check-diag n M = *list-ex* (λ *i*. *op-mtx-get M* (*i*, *i*) < *Le 0*) [*0*..<*Suc n*]
unfolding *check-diag-def list-ex-iff* **by** *fastforce*

definition *dbm-subset* :: *nat* ⇒ (*′t* :: {*linorder*, *zero*}) *DBM′* ⇒ *′t DBM′* ⇒ *bool* **where**
dbm-subset n M M′ ≡ *check-diag n M* ∨ *pointwise-cmp* (≤) *n* (*curry M*) (*curry M′*)

lemma *dbm-subset-refl*:
dbm-subset n M M
unfolding *dbm-subset-def pointwise-cmp-def* **by** *simp*

lemma *dbm-subset-trans*:
assumes *dbm-subset n M1 M2 dbm-subset n M2 M3*
shows *dbm-subset n M1 M3*
using *assms* **unfolding** *dbm-subset-def pointwise-cmp-def check-diag-def* **by** *fastforce*

lemma *canonical-nonneg-diag-non-empty*:
assumes *canonical M n* $\forall i \le n.\ 0 \le M\ i\ i$ $\forall c.\ v\ c \le n \longrightarrow 0 < v\ c$

shows $[M]_{v,n} \neq \{\}$
apply (*rule cyc-free-not-empty*)
apply (*rule canonical-cyc-free*)
using *assms* **by** *auto*

The type constraint in this lemma is due to $[\![canonical\ ?M\ ?n;\ [?M]_{?v,?n} \subseteq [?M']_{?v,?n};\ [?M]_{?v,?n} \neq \{\};\ ?i \leq ?n;\ ?j \leq ?n;\ ?i \neq ?j;\ \forall c.\ 0 < ?v\ c \wedge (\forall x\ y.\ ?v\ x \leq ?n \wedge ?v\ y \leq ?n \wedge ?v\ x = ?v\ y \longrightarrow x = y);\ \forall k \leq ?n.\ 0 < k \longrightarrow (\exists c.\ ?v\ c = k)]\!] \Longrightarrow ?M\ ?i\ ?j \leq ?M'\ ?i\ ?j$. Proving it for a more general class of types is possible but also tricky due to a missing setup for arithmetic.

lemma *subset-eq-dbm-subset*:
fixes M :: *real DBM'*
assumes *canonical* (*curry M*) *n* ∨ *check-diag n M* $\forall\ i \leq n.\ M\ (i,\ i) \leq 0$ $\forall\ i \leq n.\ M'\ (i,\ i) \leq 0$
and *cn*: *clock-numbering' v n* **and** *surj*: $\forall\ k \leq n.\ 0 < k \longrightarrow (\exists c.\ v\ c = k)$
shows $[curry\ M]_{v,n} \subseteq [curry\ M']_{v,n} \longleftrightarrow$ *dbm-subset n M M'*
proof (*cases check-diag n M*)
case *True*
with *check-diag-empty*[*OF surj*] **show** *?thesis* **unfolding** *dbm-subset-def* **by** *auto*
next
case *F*: *False*
with *assms*(*1*) **have** *canonical*: *canonical* (*curry M*) *n* **by** *fast*
show *?thesis*
proof (*cases check-diag n M'*)
case *True*
from *F cn* **have**
$[curry\ M]_{v,n} \neq \{\}$
apply −
apply (*rule canonical-nonneg-diag-non-empty*[*OF canonical*])
unfolding *check-diag-def neutral*[*symmetric*] **by** *auto*
moreover from *F True* **have**
¬ *dbm-subset n M M'*
unfolding *dbm-subset-def pointwise-cmp-def check-diag-def* **by** *fastforce*
ultimately show *?thesis* **using** *check-diag-empty*[*OF surj True*] **by** *auto*
next
case *False*
with *F assms*(*2*,*3*) **have**
$\forall\ i \leq n.\ M\ (i,\ i) = 0$ $\forall\ i \leq n.\ M'\ (i,\ i) = 0$
unfolding *check-diag-def neutral*[*symmetric*] **by** *fastforce+*
with *F False* **show** *?thesis* **unfolding** *dbm-subset-def*
by (*subst subset-eq-pointwise-le*[*OF canonical - - cn surj*]; *auto*)

qed
qed

lemma *pointwise-cmp-alt-def*:
pointwise-cmp P n M M′ =
list-all (λ *i*. *list-all* (λ *j*. *P* (*M i j*) (*M′ i j*)) [*0*..<*Suc n*]) [*0*..<*Suc n*]
unfolding *pointwise-cmp-def* **by** (*fastforce simp*: *list-all-iff*)

lemma *dbm-subset-alt-def*[*code*]:
dbm-subset n M M′ =
(*list-ex* (λ *i*. *op-mtx-get M* (*i*, *i*) < *Le 0*) [*0*..<*Suc n*] ∨
list-all (λ *i*. *list-all* (λ *j*. (*op-mtx-get M* (*i*, *j*) ≤ *op-mtx-get M′* (*i*, *j*)))
[*0*..<*Suc n*]) [*0*..<*Suc n*])
by (*simp add*: *dbm-subset-def check-diag-alt-def pointwise-cmp-alt-def*)

definition *pointwise-cmp-alt-def* **where**
pointwise-cmp-alt-def P n M M′ = *fold* (λ *i b*. *fold* (λ *j b*. *P* (*M i j*) (*M′ i j*) ∧ *b*) [*1*..<*Suc n*] *b*) [*1*..<*Suc n*] *True*

lemma *list-all-foldli*:
list-all P xs = *foldli xs* (λ*x*. *x* = *True*) (λ *x* -. *P x*) *True*
apply (*induction xs*)
apply (*simp*; *fail*)
subgoal for *x xs*
apply *simp*
apply (*induction xs*)
by *auto*
done

lemma *list-ex-foldli*:
list-ex P xs = *foldli xs Not* (λ *x y*. *P x* ∨ *y*) *False*
apply (*induction xs*)
apply (*simp*; *fail*)
subgoal for *x xs*
apply *simp*
apply (*induction xs*)
by *auto*
done

5.6 Extrapolations

context
fixes
upd-entry :: *nat* ⇒ *nat* ⇒ *′t* ⇒ *′t* ⇒ (*′t*::{*linordered-ab-group-add*})

DBMEntry ⇒ *'t DBMEntry*
  and *upd-entry-0* :: *nat* ⇒ *'t* ⇒ *'t DBMEntry* ⇒ *'t DBMEntry*
begin

definition *extra* ::
  't DBM ⇒ (*nat* ⇒ *'t*) ⇒ (*nat* ⇒ *'t*) ⇒ *nat* ⇒ *'t DBM*
where
  extra M l u n ≡ *λi j.*
    let ub = if i > 0 then l i else 0 in
    let lb = if j > 0 then u j else 0 in
    if i ≤ n ∧ j ≤ n then
      if i ≠ j then
        if i > 0 then upd-entry i j lb ub (*M i j*) *else upd-entry-0 j lb* (*M i j*)
      else norm-diag (*M i j*)
    else M i j

definition *upd-line-0* ::
  't DBM' ⇒ *'t list* ⇒ *nat* ⇒ *'t DBM'*
where
  upd-line-0 M k n =
    fold
      (*λj M.*
        M((*0*, *j*) := *upd-entry-0 j* (*op-list-get k j*) (*M*(*0*, *j*))))
      [*1..<Suc n*]
      (*M*((*0*, *0*) := *norm-diag* (*M* (*0*, *0*))))

definition *upd-line* ::
  't DBM' ⇒ *'t list* ⇒ *'t* ⇒ *nat* ⇒ *nat* ⇒ *'t DBM'*
where
  upd-line M k ub i n =
    fold
      (*λj M.*
        if i ≠ j then
          M((*i*, *j*) := *upd-entry i j* (*op-list-get k j*) *ub* (*M*(*i*, *j*)))
        else M((*i*, *j*) := *norm-diag* (*M* (*i*, *j*))))
      [*1..<Suc n*]
      (*M*((*i*, *0*) := *upd-entry i 0 0 ub* (*M*(*i*, *0*))))

lemma *upd-line-Suc-unfold*:
  upd-line M k ub i (*Suc n*) = (*let M'* = *upd-line M k ub i n in*
  if i ≠ Suc n then
    M' ((*i*, *Suc n*) := *upd-entry i* (*Suc n*) (*op-list-get k* (*Suc n*)) *ub* (*M'*(*i*,
Suc n)))
  else M' ((*i*, *Suc n*) := *norm-diag* (*M'* (*i*, *Suc n*))))

unfolding *upd-line-def* **by** *simp*

lemma *upd-line-out-of-bounds*:
  assumes $j > n$
  shows *upd-line M k ub i n* $(i', j) = M (i', j)$
  using *assms* **by** (*induction n*) (*auto simp*: *upd-line-def*)

lemma *upd-line-alt-def*:
  assumes $i > 0$
  shows
  upd-line M k ub i n $(i', j) = ($
    let lb = *if* $j > 0$ *then op-list-get k j else 0 in*
    if $i' = i \wedge j \leq n$ *then*
      if $i \neq j$ *then*
        upd-entry i j lb ub $(M (i, j))$
      else
        norm-diag $(M (i, j))$
    else $M (i', j)$
)
  using *assms*
  apply *simp*
  apply *safe*
    apply (*induction n*, *simp add*: *upd-line-def*,
    auto simp: *upd-line-out-of-bounds upd-line-Suc-unfold Let-def*)+
  done

lemma *upd-line-0-alt-def*:
  upd-line-0 M k n $(i', j) = ($
    if $i' = 0 \wedge j \leq n$ *then*
      if $j > 0$ *then upd-entry-0 j* (*op-list-get k j*) $(M (0, j))$ *else norm-diag* $(M (0, 0))$
    else $M (i', j)$
)
  by (*induction n*) (*auto simp*: *upd-line-0-def*)

definition *extra-upd* :: *'t DBM'* ⇒ *'t list* ⇒ *'t list* ⇒ *nat* ⇒ *'t DBM'*
where
  extra-upd M l u n ≡
    fold (λ*i M*. *upd-line M u* (*op-list-get l i*) *i n*) [*1..<Suc n*] (*upd-line-0 M u n*)

lemma *upd-line-0-out-ouf-bounds1*:
  assumes $i > 0$
  shows *upd-line-0 M k n* $(i, j) = M (i, j)$

using *assms* **unfolding** *upd-line-0-alt-def* **by** *simp*

lemma *upd-line-0-out-ouf-bounds2*:
assumes $j > n$
shows *upd-line-0 M k n* $(i, j) = M\ (i, j)$
using *assms* **unfolding** *upd-line-0-alt-def* **by** *simp*

lemma *upd-out-of-bounds-aux1*:
assumes $i > n$
shows *fold* (λi *M*. *upd-line M k* (*op-list-get l i*) *i m*) [*1..<Suc n*] $M\ (i, j)$ $= M\ (i, j)$
using *assms*
by (*intro fold-invariant*[**where** $Q = \lambda i.\ i > 0 \wedge i \leq n$ **and** $P = \lambda M'.\ M'\ (i, j) = M\ (i, j)$])
(*auto simp*: *upd-line-alt-def*)

lemma *upd-out-of-bounds-aux2*:
assumes $j > m$
shows *fold* (λi *M*. *upd-line M k* (*op-list-get l i*) *i m*) [*1..<Suc n*] $M\ (i, j)$ $= M\ (i, j)$
using *assms*
by (*intro fold-invariant*[**where** $Q = \lambda i.\ i > 0 \wedge i \leq n$ **and** $P = \lambda M'.\ M'\ (i, j) = M\ (i, j)$])
(*auto simp*: *upd-line-alt-def*)

lemma *upd-out-of-bounds1*:
assumes $i > n$
shows *extra-upd M l u n* $(i, j) = M\ (i, j)$
using *assms* **unfolding** *extra-upd-def*
by (*subst upd-out-of-bounds-aux1*) (*auto simp*: *upd-line-0-out-ouf-bounds1*)

lemma *upd-out-of-bounds2*:
assumes $j > n$
shows *extra-upd M l u n* $(i, j) = M\ (i, j)$
by (*simp only*: *assms extra-upd-def upd-out-of-bounds-aux2 upd-line-0-out-ouf-bounds2*)

definition *norm-entry* **where**
norm-entry x l u i j = (
let ub = *if* $i > 0$ *then* (*l* ! *i*) *else 0 in*
let lb = *if* $j > 0$ *then* (*u* ! *j*) *else 0 in*
if $i \neq j$ *then if* $i = 0$ *then upd-entry-0 j lb x else upd-entry i j lb ub x else norm-diag x*)

lemma *upd-extra-aux*:

assumes $i \le n$ $j \le m$
shows
fold (λi *M*. *upd-line M u* (*op-list-get l i*) *i m*) [*1..<Suc n*] (*upd-line-0 M u m*) (*i*, *j*)
= *norm-entry* (*M* (*i*, *j*)) *l u i j*
using *assms upd-out-of-bounds-aux1*[*unfolded op-list-get-def*]
apply (*induction n*)
apply (*simp add*: *upd-line-0-alt-def norm-entry-def*; *fail*)
apply (*auto simp*: *upd-line-alt-def upt-Suc-append upd-line-0-out-ouf-bounds1 norm-entry-def*
simp del: *upt-Suc*)
done

lemma *upd-extra-aux'*:
assumes $i \le n$ $j \le n$
shows *extra-upd M l u n* (*i*, *j*) = *extra* (*curry M*) (λi. *l* ! *i*) (λi. *u* ! *i*) *n i j*
using *assms* **unfolding** *extra-upd-def*
by (*subst upd-extra-aux*[*OF assms*]) (*simp add*: *norm-entry-def extra-def norm-diag-def Let-def*)

lemma *extra-upd-extra''*:
extra-upd M l u n (*i*, *j*) = *extra* (*curry M*) (λi. *l* ! *i*) (λi. *u* ! *i*) *n i j*
by (*cases* $i > n$; *cases* $j > n$;
simp add: *upd-out-of-bounds1 upd-out-of-bounds2 extra-def upd-extra-aux'*)

lemma *extra-upd-extra'*:
curry (*extra-upd M l u n*) = *extra* (*curry M*) (λi. *l* ! *i*) (λi. *u* ! *i*) *n*
by (*simp add*: *curry-def extra-upd-extra''*)

lemma *extra-upd-extra*:
extra-upd = (λM *l u n* (*i*, *j*). *extra* (*curry M*) (λi. *l* ! *i*) (λi. *u* ! *i*) *n i j*)
by (*intro ext*) (*clarsimp simp*: *extra-upd-extra''*)

end

lemma *norm-is-extra*:
norm M k n =
extra
(λ- - *lb ub e*. *norm-lower* (*norm-upper e ub*) (−*lb*))
(λ- *lb e*. *norm-lower* (*norm-upper e 0*) (−*lb*)) *M k k n*
unfolding *norm-def extra-def Let-def* **by** (*intro ext*) *auto*

lemma *extra-lu-is-extra*:

extra-lu M l u n =
extra
(λ- - *lb ub e. norm-lower* (*norm-upper e ub*) ($-$*lb*))
(λ- *lb e. norm-lower* (*norm-upper e 0*) ($-$*lb*)) *M l u n*
unfolding *extra-def extra-lu-def Let-def* **by** (*intro ext*) *auto*

lemma *extra-lup-is-extra*:
extra-lup M l u n =
extra
(λ*i j lb ub e. if Lt ub* $\prec$ *e then* ∞
else if M 0 i $\prec$ *Lt* ($-$ *ub*) *then* ∞
else if M 0 j $\prec$ (*if j* $>$ *0 then Lt* ($-$ *lb*) *else Lt 0*) *then* ∞
else e)
(λ*j lb e. if Le 0* $\prec$ *M 0 j then* ∞
else if M 0 j $\prec$ (*if j* $>$ *0 then Lt* ($-$ *lb*) *else Lt 0*) *then Lt* ($-$ *lb*)
else M 0 j) *M l u n*
unfolding *extra-def extra-lup-def Let-def* **by** (*intro ext*) *auto*

definition
norm-upd M k =
extra-upd
(λ- - *lb ub e. norm-lower* (*norm-upper e ub*) ($-$*lb*))
(λ- *lb e. norm-lower* (*norm-upper e 0*) ($-$*lb*)) *M k k*

definition
extra-lu-upd =
extra-upd
(λ- - *lb ub e. norm-lower* (*norm-upper e ub*) ($-$*lb*))
(λ- *lb e. norm-lower* (*norm-upper e 0*) ($-$*lb*))

definition
extra-lup-upd M =
extra-upd
(λ*i j lb ub e. if Lt ub* $\prec$ *e then* ∞
else if M (*0*, *i*) $\prec$ *Lt* ($-$ *ub*) *then* ∞
else if M (*0*, *j*) $\prec$ (*if j* $>$ *0 then Lt* ($-$ *lb*) *else Lt 0*) *then* ∞
else e)
(λ*j lb e. if Le 0* $\prec$ *M* (*0*, *j*) *then* ∞
else if M (*0*, *j*) $\prec$ (*if j* $>$ *0 then Lt* ($-$ *lb*) *else Lt 0*) *then Lt* ($-$ *lb*)
else M (*0*, *j*)) *M*

lemma *extra-upd-cong*:
assumes $\bigwedge$*i j x y e.* $i \le n \Longrightarrow j \le n \Longrightarrow$ *upd-entry i j x y e* = *upd-entry'* *i j x y e*

$\bigwedge$*i x e. i* $\leq$ *n* $\Longrightarrow$ *upd-entry-0 i x e* = *upd-entry-0′ i x e*
shows *extra-upd upd-entry upd-entry-0 M l u n* = *extra-upd upd-entry′ upd-entry-0′ M l u n*
unfolding *extra-upd-def upd-line-def upd-line-0-def*
apply (*intro fold-cong*)
apply (*auto simp*: *assms*)[*4*]
apply (*rule ext*, *rule fold-cong*, *auto simp*: *assms*)
done

lemma *extra-lup-upd-alt-def*:
extra-lup-upd M l u n = (
let xs = *IArray* (*map* (λ*i. M* (*0*, *i*)) [*0..*<*Suc n*]) *in*
extra-upd
(λ*i j lb ub e. if Lt ub* $\prec$ *e then* ∞
else if (*xs* !! *i*) $\prec$ *Lt* (− *ub*) *then* ∞
else if (*xs* !! *j*) $\prec$ (*if j* > *0 then Lt* (− *lb*) *else Lt 0*) *then* ∞
else e)
(λ*j lb e. if Le 0* $\prec$ (*xs* !! *j*) *then* ∞
else if (*xs* !! *j*) $\prec$ (*if j* > *0 then Lt* (− *lb*) *else Lt 0*) *then Lt* (− *lb*)
else (*xs* !! *j*))) *M l u n*
unfolding *extra-lup-upd-def Let-def* **by** (*rule extra-upd-cong*; *clarsimp simp del*: *upt-Suc*; *fail*)

lemma *extra-lup-upd-alt-def2*:
extra-lup-upd M l u n = (
let xs = *map* (λ*i. M* (*0*, *i*)) [*0..*<*Suc n*] *in*
extra-upd
(λ*i j lb ub e. if Lt ub* $\prec$ *e then* ∞
else if (*xs* ! *i*) $\prec$ *Lt* (− *ub*) *then* ∞
else if (*xs* ! *j*) $\prec$ (*if j* > *0 then Lt* (− *lb*) *else Lt 0*) *then* ∞
else e)
(λ*j lb e. if Le 0* $\prec$ (*xs* ! *j*) *then* ∞
else if (*xs* ! *j*) $\prec$ (*if j* > *0 then Lt* (− *lb*) *else Lt 0*) *then Lt* (− *lb*)
else (*xs* ! *j*)) *M l u n*)
unfolding *extra-lup-upd-def Let-def* **by** (*rule extra-upd-cong*; *clarsimp simp del*: *upt-Suc*; *fail*)

lemma *norm-upd-norm*: *norm-upd* = (λ*M k n* (*i*, *j*). *norm* (*curry M*) (λ*i. k* ! *i*) *n i j*)
and *extra-lu-upd-extra-lu*:
extra-lu-upd = (λ*M l u n* (*i*, *j*). *extra-lu* (*curry M*) (λ*i. l* ! *i*) (λ*i. u* ! *i*) *n i j*)
and *extra-lup-upd-extra-lup*:
extra-lup-upd = (λ*M l u n* (*i*, *j*). *extra-lup* (*curry M*) (λ*i. l* ! *i*) (λ*i. u* !

i) *n i j*)
unfolding *norm-upd-def norm-is-extra extra-lu-upd-def extra-lu-is-extra extra-lup-upd-def extra-lup-is-extra extra-upd-extra curry-def*
by *standard*+

lemma *norm-upd-norm′*:
curry (*norm-upd M k n*) = *norm* (*curry M*) (*λi. k* ! *i*) *n*
unfolding *norm-upd-norm* **by** *simp*

— Copy from Regions Beta, original should be moved
lemma *norm-int-preservation*:
assumes *dbm-int M n* ∀ *c* ≤ *n*. *k c* ∈ ℤ
shows *dbm-int* (*norm M k n*) *n*
using *assms* **unfolding** *norm-def norm-diag-def* **by** (*auto simp*: *Let-def*)

lemma
assumes *dbm-int M n* ∀ *c* ≤ *n*. *l c* ∈ ℤ ∀ *c* ≤ *n*. *u c* ∈ ℤ
shows *extra-lu-preservation*: *dbm-int* (*extra-lu M l u n*) *n*
and *extra-lup-preservation*: *dbm-int* (*extra-lup M l u n*) *n*
using *assms* **unfolding** *extra-lu-def extra-lup-def norm-diag-def* **by** (*auto simp*: *Let-def*)

lemma *norm-upd-int-preservation*:
fixes *M* :: (′*t* :: {*linordered-ab-group-add*, *ring-1*}) *DBM*′
assumes *dbm-int* (*curry M*) *n* ∀ *c* ∈ *set k*. *c* ∈ ℤ *length k* = *Suc n*
shows *dbm-int* (*curry* (*norm-upd M k n*)) *n*
using *norm-int-preservation*[*OF assms*(*1*)] *assms*(*2*,*3*) **unfolding** *norm-upd-norm curry-def* **by** *simp*

lemma
fixes *M* :: (′*t* :: {*linordered-ab-group-add*, *ring-1*}) *DBM*′
assumes *dbm-int* (*curry M*) *n*
∀ *c* ∈ *set l*. *c* ∈ ℤ *length l* = *Suc n* ∀ *c* ∈ *set u*. *c* ∈ ℤ *length u* = *Suc n*
shows *extra-lu-upd-int-preservation*: *dbm-int* (*curry* (*extra-lu-upd M l u n*)) *n*
and *extra-lup-upd-int-preservation*: *dbm-int* (*curry* (*extra-lup-upd M l u n*)) *n*
using *extra-lu-preservation*[*OF assms*(*1*)] *extra-lup-preservation*[*OF assms*(*1*)] *assms*(*2*−)
unfolding *extra-lu-upd-extra-lu extra-lup-upd-extra-lup curry-def* **by** *simp*+

lemma
assumes *dbm-default* (*curry M*) *n*
shows *norm-upd-default*: *dbm-default* (*curry* (*norm-upd M k n*)) *n*

and *extra-lu-upd-default*: *dbm-default* (*curry* (*extra-lu-upd M l u n*)) *n*
and *extra-lup-upd-default*: *dbm-default* (*curry* (*extra-lup-upd M l u n*)) *n*
using *assms* **unfolding**
norm-upd-norm norm-def extra-lu-upd-extra-lu extra-lu-def extra-lup-upd-extra-lup extra-lup-def
by *auto*

end
theory *DBM-Imperative-Loops*
imports
Refine-Imperative-HOL.IICF
begin

5.6.1 Additional proof rules for typical looping constructs

Heap-Monad.fold-map **lemma** *fold-map-ht*:
assumes *list-all* (λ*x*. <*A* ∗ *true*> *f x* <λ*r*. ↑(*Q x r*) ∗ *A*>$_t$) *xs*
shows <*A* ∗ *true*> *Heap-Monad.fold-map f xs* <λ*rs*. ↑(*list-all2* (λ*x r*. *Q x r*) *xs rs*) ∗ *A*>$_t$
using *assms* **by** (*induction xs*; *sep-auto*)

lemma *fold-map-ht*′:
assumes *list-all* (λ*x*. <*true*> *f x* <λ*r*. ↑(*Q x r*)>$_t$) *xs*
shows <*true*> *Heap-Monad.fold-map f xs* <λ*rs*. ↑(*list-all2* (λ*x r*. *Q x r*) *xs rs*)>$_t$
using *assms* **by** (*induction xs*; *sep-auto*)

lemma *fold-map-ht1*:
assumes ⋀*x xi*. <*A* ∗ *R x xi* ∗ *true*> *f xi* <λ*r*. *A* ∗ ↑(*Q x r*)>$_t$
shows
<*A* ∗ *list-assn R xs xsi* ∗ *true*>
Heap-Monad.fold-map f xsi
<λ*rs*. *A* ∗ ↑(*list-all2* (λ*x r*. *Q x r*) *xs rs*)>$_t$
apply (*induction xs arbitrary*: *xsi*)
apply (*sep-auto*; *fail*)
subgoal for *x xs xsi*
by (*cases xsi*; *sep-auto heap*: *assms*)
done

lemma *fold-map-ht2*:
assumes ⋀*x xi*. <*A* ∗ *R x xi* ∗ *true*> *f xi* <λ*r*. *A* ∗ *R x xi* ∗ ↑(*Q x r*)>$_t$
shows
<*A* ∗ *list-assn R xs xsi* ∗ *true*>

Heap-Monad.fold-map f xsi
$<\lambda rs.\ A * \textit{list-assn}\ R\ xs\ xsi * \uparrow(\textit{list-all2}\ (\lambda x\ r.\ Q\ x\ r)\ xs\ rs)>_t$
apply (*induction xs arbitrary*: *xsi*)
apply (*sep-auto*; *fail*)
subgoal for *x xs xsi*
apply (*cases xsi*; *sep-auto heap*: *assms*)
apply (*rule cons-rule*[*rotated 2*], *rule frame-rule*, *rprems*)
apply *frame-inference*
apply *frame-inference*
apply *sep-auto*
done
done

lemma *fold-map-ht3*:
assumes $\bigwedge x\ xi.\ <A * R\ x\ xi * \textit{true}>\ f\ xi\ <\lambda r.\ A * Q\ x\ r>_t$
shows $<A * \textit{list-assn}\ R\ xs\ xsi * \textit{true}>$ *Heap-Monad.fold-map f xsi* $<\lambda rs.\ A * \textit{list-assn}\ Q\ xs\ rs>_t$
apply (*induction xs arbitrary*: *xsi*)
apply (*sep-auto*; *fail*)
subgoal for *x xs xsi*
apply (*cases xsi*; *sep-auto heap*: *assms*)
apply (*rule Hoare-Triple.cons-pre-rule*[*rotated*], *rule frame-rule*, *rprems*, *frame-inference*)
apply *sep-auto*
done
done

imp-for′ **and** *imp-for* **lemma** *imp-for-rule2*:
assumes
$\textit{emp} \Longrightarrow_A I\ i\ a$
$\bigwedge i\ a.\ <A * I\ i\ a * \textit{true}>\ ci\ a\ <\lambda r.\ A * I\ i\ a * \uparrow(r \longleftrightarrow c\ a)>_t$
$\bigwedge i\ a.\ i < j \Longrightarrow c\ a \Longrightarrow <A * I\ i\ a * \textit{true}>\ f\ i\ a\ <\lambda r.\ A * I\ (i + 1)\ r>_t$
$\bigwedge a.\ I\ j\ a \Longrightarrow_A Q\ a\ \bigwedge i\ a.\ i < j \Longrightarrow \neg\ c\ a \Longrightarrow I\ i\ a \Longrightarrow_A Q\ a$
$i \le j$
shows $<A * \textit{true}>\ \textit{imp-for}\ i\ j\ ci\ f\ a\ <\lambda r.\ A * Q\ r>_t$
proof −
have
$<A * I\ i\ a * \textit{true}>$
imp-for i j ci f a
$<\lambda r.\ A * (I\ j\ r \vee_A (\exists_A\ i'.\ \uparrow(i' < j \wedge \neg\ c\ r) * I\ i'\ r))>_t$
using ‹$i \le j$› *assms*(*2*,*3*)
apply (*induction* $j - i$ *arbitrary*: *i a*; *sep-auto*)

```
    subgoal
      apply (rule ent-star-mono, rule ent-star-mono)
        apply (rule ent-refl, rule ent-disjI1-direct, rule ent-refl)
      done
     apply rprems
    apply sep-auto
      apply (rprems)
       apply sep-auto+
   apply (rule ent-star-mono, rule ent-star-mono, rule ent-refl, rule ent-disjI2')
     apply solve-entails
     apply simp+
    done
  then show ?thesis
    apply (rule cons-rule[rotated 2])
    subgoal
      apply (subst merge-true-star[symmetric])
      apply (rule ent-frame-fwd[OF assms(1)])
       apply frame-inference+
      done
    apply (rule ent-star-mono)
     apply (rule ent-star-mono, rule ent-refl)
     apply (solve-entails eintros: assms(5) assms(4) ent-disjE)+
    done
qed

lemma imp-for-rule:
  assumes
    emp ⟹A I i a
    ⋀i a. <I i a * true> ci a <λr. I i a * ↑(r ⟷ c a)>t
    ⋀i a. i < j ⟹ c a ⟹ <I i a * true> f i a <λr. I (i + 1) r>t
    ⋀a. I j a ⟹A Q a ⋀i a. i < j ⟹ ¬ c a ⟹ I i a ⟹A Q a
    i ≤ j
  shows <true> imp-for i j ci f a <λr. Q r>t
  by (rule cons-rule[rotated 2], rule imp-for-rule2[where A = true])
     (rule assms | sep-auto heap: assms; fail)+

lemma imp-for'-rule2:
  assumes
    emp ⟹A I i a
    ⋀i a. i < j ⟹ <A * I i a * true> f i a <λr. A * I (i + 1) r>t
    ⋀a. I j a ⟹A Q a
    i ≤ j
  shows <A * true> imp-for' i j f a <λr. A * Q r>t
  unfolding imp-for-imp-for'[symmetric] using assms(3,4)
```

by (*sep-auto heap*: *assms imp-for-rule2*[**where** *c* = λ-. *True*])

lemma *imp-for′-rule*:
assumes
emp ⟹$_A$ *I i a*
⋀*i a*. *i* < *j* ⟹ <*I i a* ∗ *true*> *f i a* <λ*r*. *I* (*i* + *1*) *r*>$_t$
⋀*a*. *I j a* ⟹$_A$ *Q a*
i ≤ *j*
shows <*true*> *imp-for′ i j f a* <λ*r*. *Q r*>$_t$
unfolding *imp-for-imp-for′*[*symmetric*] **using** *assms*(*3*,*4*)
by (*sep-auto heap*: *assms imp-for-rule*[**where** *c* = λ-. *True*])

lemma *nth-rule*:
assumes *is-pure S*
and *b* < *length a*
shows
<*nat-assn b bi* ∗ *array-assn S a ai*> *Array.nth ai bi*
<λ*r*. ∃$_A$*x*. *nat-assn b bi* ∗ *array-assn S a ai* ∗ *S x r* ∗ *true* ∗ ↑ (*x* = *a* ! *b*)>
using *sepref-fr-rules*(*165*)[*unfolded hn-refine-def hn-ctxt-def*] *assms* **by** *force*

lemma *imp-for-list-all*:
assumes
is-pure R n ≤ *length xs*
⋀*x xi*. <*A* ∗ *R x xi* ∗ *true*> *Pi xi* <λ*r*. *A* ∗ ↑ (*r* ⟷ *P x*)>$_t$
shows
<*A* ∗ *array-assn R xs a* ∗ *true*>
imp-for 0 n Heap-Monad.return
(λ*i* -. *do* {
x ← *Array.nth a i*; *Pi x*
})
True
<λ*r*. *A* ∗ *array-assn R xs a* ∗ ↑(*r* ⟷ *list-all P* (*take n xs*))>$_t$
apply (*rule imp-for-rule2*[**where** *I* = λ*i r*. ↑ (*r* ⟷ *list-all P* (*take i xs*))])
apply *sep-auto*
apply *sep-auto*
subgoal for *i b*
using *assms*(*2*)
apply (*sep-auto heap*: *nth-rule*)
apply (*rule cons-rule*[*rotated 2*], *rule frame-rule*,
rule nth-rule[**where** *b* = *i* **and** *a* = *xs*], *rule assms*)
apply *simp*

apply (*simp add*: *pure-def*)
apply *frame-inference*
apply *frame-inference*
apply (*sep-auto heap*: *assms*(*3*) *simp*: *pure-def take-Suc-conv-app-nth*)
done
apply (*simp add*: *take-Suc-conv-app-nth*)
apply *simp*
unfolding *list-all-iff*
apply *clarsimp*
apply (*metis le-less set-take-subset-set-take subsetCE*)
..

lemma *imp-for-list-ex*:
assumes
is-pure R n ≤ *length xs*
⋀*x xi*. <*A* ∗ *R x xi* ∗ *true*> *Pi xi* <λ*r*. *A* ∗ ↑ (*r* ⟷ *P x*)>$_t$
shows
<*A* ∗ *array-assn R xs a* ∗ *true*>
imp-for 0 n (λ*x*. *Heap-Monad.return* (¬ *x*))
(λ*i* -. *do* {
x ← *Array.nth a i*; *Pi x*
})
False
<λ*r*. *A* ∗ *array-assn R xs a* ∗ ↑(*r* ⟷ *list-ex P* (*take n xs*))>$_t$
apply (*rule imp-for-rule2*[**where** *I* = λ*i r*. ↑ (*r* ⟷ *list-ex P* (*take i xs*))])
apply *sep-auto*
apply *sep-auto*
subgoal for *i b*
using *assms*(*2*)
apply (*sep-auto heap*: *nth-rule*)
apply (*rule cons-rule*[*rotated 2*], *rule frame-rule*, *rule nth-rule*[*of - i xs*], *rule assms*)
apply *simp*
apply (*simp add*: *pure-def*)
apply *frame-inference*
apply *frame-inference*
apply (*sep-auto heap*: *assms*(*3*) *simp*: *pure-def take-Suc-conv-app-nth*)
done
apply (*simp add*: *take-Suc-conv-app-nth*)
apply *simp*
unfolding *list-ex-iff*
apply *clarsimp*
apply (*metis le-less set-take-subset-set-take subsetCE*)

..

```
lemma imp-for-list-all2:
  assumes
    is-pure R is-pure S n ≤ length xs n ≤ length ys
    ⋀x xi y yi. <A ∗ R x xi ∗ S y yi ∗ true> Pi xi yi <λr. A ∗ ↑ (r ⟷ P
x y)>t
  shows
  <A ∗ array-assn R xs a  ∗ array-assn S ys b ∗ true>
    imp-for 0 n Heap-Monad.return
    (λi -. do {
      x ← Array.nth a i; y ← Array.nth b i; Pi x y
    })
    True
  <λr. A ∗  array-assn R xs a  ∗ array-assn S ys b ∗ ↑(r ⟷ list-all2 P
(take n xs) (take n ys))>t
  apply (rule imp-for-rule2[where I = λi r. ↑ (r ⟷ list-all2 P (take i
xs) (take i ys))])
       apply (sep-auto; fail)
      apply (sep-auto; fail)
  subgoal for i -
    supply [simp] = pure-def
    using assms(3,4)
    apply sep-auto
     apply (rule cons-rule[rotated 2], rule frame-rule, rule nth-rule[of - i xs],
rule assms)
       apply force
      apply (simp, frame-inference; fail)
     apply frame-inference
    apply sep-auto

     apply (rule cons-rule[rotated 2], rule frame-rule, rule nth-rule[of - i ys],
rule assms)
    unfolding pure-def
       apply force
      apply (simp, frame-inference; fail)
     apply frame-inference
    apply sep-auto

    supply [sep-heap-rules] = assms(5)
    apply sep-auto
    subgoal
      unfolding list-all2-conv-all-nth apply clarsimp
      subgoal for i′
```

by (*cases* $i' = i$) *auto*
done
subgoal
unfolding *list-all2-conv-all-nth* **by** *clarsimp*
apply *frame-inference*
done
unfolding *list-all2-conv-all-nth* **apply** *auto*
done

lemma *imp-for-list-all2'*:
assumes
is-pure R is-pure S $n \leq$ *length xs* $n \leq$ *length ys*
$\bigwedge$*x xi y yi*. <*R x xi* $*$ *S y yi*> *Pi xi yi* <λr. $\uparrow$ ($r \longleftrightarrow$ *P x y*)>$_t$
shows
<*array-assn R xs a* $*$ *array-assn S ys b*>
imp-for 0 n Heap-Monad.return
(λi -. *do* {
$x \leftarrow$ *Array.nth a i*; $y \leftarrow$ *Array.nth b i*; *Pi x y*
})
True
<λr. *array-assn R xs a* $*$ *array-assn S ys b* $*$ $\uparrow$($r \longleftrightarrow$ *list-all2 P* (*take n xs*) (*take n ys*))>$_t$
by (*rule cons-rule*[*rotated 2*], *rule imp-for-list-all2*[**where** *A* = *true*, *rotated 4*])
(*sep-auto heap*: *assms intro*: *assms*)+

end
theory *DBM-Operations-Impl-Refine*
imports
DBM-Operations-Impl
HOL−Library.IArray
DBM-Imperative-Loops
begin

lemma *rev-map-fold-append-aux*:
fold (λ *x xs*. *f x* # *xs*) *xs zs* @ *ys* = *fold* (λ *x xs*. *f x* # *xs*) *xs* (*zs*@*ys*)
by (*induction xs arbitrary*: *zs*) *auto*

lemma *rev-map-fold*:
rev (*map f xs*) = *fold* (λ *x xs*. *f x* # *xs*) *xs* []
by (*induction xs*; *simp add*: *rev-map-fold-append-aux*)

lemma *map-rev-fold*:
map f xs = *rev* (*fold* (λ *x xs*. *f x* # *xs*) *xs* [])

```
  using rev-map-fold rev-swap by fastforce

lemma pointwise-cmp-iff:
  pointwise-cmp P n M M' ⟷ list-all2 P (take ((n + 1) * (n + 1)) xs)
(take ((n + 1) * (n + 1)) ys)
  if ∀ i≤n. ∀ j≤n. xs ! (i + i * n + j) = M i j
    ∀ i≤n. ∀ j≤n. ys ! (i + i * n + j) = M' i j
    (n + 1) * (n + 1) ≤ length xs (n + 1) * (n + 1) ≤ length ys
  using that unfolding pointwise-cmp-def
  unfolding list-all2-conv-all-nth
  apply clarsimp
  apply safe
  subgoal premises prems for x
  proof −
    let ?i = x div (n + 1) let ?j = x mod (n + 1)
    from ‹x < -› have ?i < Suc n ?j ≤n
      by (simp add: less-mult-imp-div-less)+
    with prems have
      xs ! (?i + ?i * n + ?j) = M ?i ?j ys ! (?i + ?i * n + ?j) = M' ?i ?j
      P (M ?i ?j) (M' ?i ?j)
      by auto
    moreover have ?i + ?i * n + ?j = x
    by (metis ab-semigroup-add-class.add.commute mod-div-mult-eq mult-Suc-right
plus-1-eq-Suc)
    ultimately show ‹P (xs ! x) (ys ! x)›
      by auto
  qed
  subgoal for i j
    apply (erule allE[of - i], erule impE, simp)
    apply (erule allE[of - i], erule impE, simp)
    apply (erule allE[of - i + i * n + j], erule impE)
    subgoal
      by (rule le-imp-less-Suc) (auto intro!: add-mono simp: algebra-simps)
    apply (erule allE[of - j], erule impE, simp)
    apply (erule allE[of - j], erule impE, simp)
    apply simp
    done
  done

fun intersperse :: 'a ⇒ 'a list ⇒ 'a list where
  intersperse sep (x # y # xs) = x # sep # intersperse sep (y # xs) |
  intersperse - xs = xs

lemma the-pure-id-assn-eq[simp]:
```

the-pure ($\lambda a\ c.\ \uparrow (c = a)$) = *Id*
proof −
have ∗: ($\lambda a\ c.\ \uparrow (c = a)$) = *pure Id*
unfolding *pure-def* **by** *simp*
show *?thesis*
by (*subst* ∗) *simp*
qed

lemma *pure-eq-conv*:
($\lambda a\ c.\ \uparrow (c = a)$) = *id-assn*
using *is-pure-assn-def is-pure-iff-pure-assn is-pure-the-pure-id-eq the-pure-id-assn-eq*
by *blast*

5.7 Refinement

instance *DBMEntry* :: ({*countable*}) *countable*
apply (*rule*
countable-classI[*of*
(λ*Le* (*a*::′*a*) ⇒ *to-nat* (*0*::*nat*,*a*) |
DBM.Lt a ⇒ *to-nat* (*1*::*nat*,*a*) |
DBM.INF ⇒ *to-nat* (*2*::*nat*,*undefined*::′*a*))])
apply (*simp split*: *DBMEntry.splits*)
done

instance *DBMEntry* :: ({*heap*}) *heap* **..**

definition *dbm-subset′* :: *nat* ⇒ (′*t* :: {*linorder*, *zero*}) *DBM′* ⇒ ′*t DBM′*
⇒ *bool* **where**
dbm-subset′ n M M′ ≡ *pointwise-cmp* (≤) *n* (*curry M*) (*curry M′*)

lemma *dbm-subset′-alt-def*:
dbm-subset′ n M M′ ≡
list-all (λ*i*. *list-all* (λ*j*. (*op-mtx-get M* (*i*, *j*) ≤ *op-mtx-get M′* (*i*, *j*)))
[*0*..<*Suc n*])
[*0*..<*Suc n*]
by (*simp add*: *dbm-subset′-def pointwise-cmp-alt-def neutral*)

lemma *dbm-subset-alt-def′*[*code*]:
dbm-subset n M M′ ⟷
list-ex (λ*i*. *op-mtx-get M* (*i*, *i*) < *0*) [*0*..<*Suc n*] ∨
list-all (λ*i*. *list-all* (λ*j*. (*op-mtx-get M* (*i*, *j*) ≤ *op-mtx-get M′* (*i*, *j*)))
[*0*..<*Suc n*])
[*0*..<*Suc n*]
by (*simp add*: *dbm-subset-def check-diag-alt-def pointwise-cmp-alt-def neu-*

tral)

definition
mtx-line-to-iarray m M = *IArray* (*map* (*λi*. *M* (*0*, *i*)) [*0*..<*Suc m*])

definition
mtx-line m (*M* :: - *DBM′*) = *map* (*λi*. *M* (*0*, *i*)) [*0*..<*Suc m*]

locale *DBM-Impl* =
fixes *n* :: *nat*
begin

abbreviation
mtx-assn :: (*nat* × *nat* ⇒ (*′a* :: {*linordered-ab-monoid-add*, *heap*})) ⇒ *′a array* ⇒ *assn*
where
mtx-assn ≡ *asmtx-assn* (*Suc n*) *id-assn*

abbreviation *clock-assn* ≡ *nbn-assn* (*Suc n*)

lemmas *Relation.IdI*[**where** *a* = ∞, *sepref-import-param*]
lemma [*sepref-import-param*]: ((+),(+)) ∈ *Id*→*Id*→*Id* **by** *simp*
lemma [*sepref-import-param*]: (*uminus*,*uminus*) ∈ (*Id*::(-∗-)*set*)→*Id* **by** *simp*
lemma [*sepref-import-param*]: (*Lt*,*Lt*) ∈ *Id*→*Id* **by** *simp*
lemma [*sepref-import-param*]: (*Le*,*Le*) ∈ *Id*→*Id* **by** *simp*
lemma [*sepref-import-param*]: (∞,∞) ∈ *Id* **by** *simp*
lemma [*sepref-import-param*]: (*min* :: - *DBMEntry* ⇒ -, *min*) ∈ *Id* → *Id* → *Id* **by** *simp*
lemma [*sepref-import-param*]: (*Suc*, *Suc*) ∈ *Id* → *Id* **by** *simp*

lemma [*sepref-import-param*]: (*norm-lower*, *norm-lower*) ∈ *Id*→*Id*→*Id* **by** *simp*
lemma [*sepref-import-param*]: (*norm-upper*, *norm-upper*) ∈ *Id*→*Id*→*Id* **by** *simp*
lemma [*sepref-import-param*]: (*norm-diag*, *norm-diag*) ∈ *Id*→*Id* **by** *simp*

end

definition *zero-clock* :: - :: *linordered-cancel-ab-monoid-add* **where**
zero-clock = *0*

sepref-register *zero-clock*

lemma [*sepref-import-param*]: (*zero-clock*, *zero-clock*) ∈ *Id* **by** *simp*

lemmas [*sepref-opt-simps*] = *zero-clock-def*

context
 fixes *n* :: *nat*
begin

interpretation *DBM-Impl n* **.**

sepref-definition *reset-canonical-upd-impl'* **is**
 uncurry2 (*uncurry* (λ*x*. *RETURN ooo reset-canonical-upd x*)) ::
 [λ(((-,*i*),*j*),-). *i*≤*n* ∧ *j*≤*n*]$_a$ *mtx-assn*^{d} $*_a$ *nat-assn*^{k} $*_a$ *nat-assn*^{k} $*_a$ *id-assn*^{k} → *mtx-assn*
 unfolding *reset-canonical-upd-alt-def op-mtx-set-def*[*symmetric*] **by** *sepref*

sepref-definition *reset-canonical-upd-impl* **is**
 uncurry2 (*uncurry* (λ*x*. *RETURN ooo reset-canonical-upd x*)) ::
 [λ(((-,*i*),*j*),-). *i*≤*n* ∧ *j*≤*n*]$_a$ *mtx-assn*^{d} $*_a$ *nat-assn*^{k} $*_a$ *nat-assn*^{k} $*_a$ *id-assn*^{k} → *mtx-assn*
 unfolding *reset-canonical-upd-alt-def op-mtx-set-def*[*symmetric*] **by** *sepref*

sepref-definition *up-canonical-upd-impl* **is**
 uncurry (*RETURN oo up-canonical-upd*) :: [λ(-,*i*). *i*≤*n*]$_a$ *mtx-assn*^{d} $*_a$ *nat-assn*^{k} → *mtx-assn*
 unfolding *up-canonical-upd-def op-mtx-set-def*[*symmetric*] **by** *sepref*

lemma [*sepref-import-param*]:
 (*Le 0*, *0*) ∈ *Id*
 unfolding *neutral* **by** *simp*

— Not sure if this is dangerous.
sepref-register *0*

sepref-definition *check-diag-impl'* **is**
 uncurry (*RETURN oo check-diag*) ::
 [λ(*i*, -). *i*≤*n*]$_a$ *nat-assn*^{k} $*_a$ *mtx-assn*^{k} → *bool-assn*
 unfolding *check-diag-alt-def list-ex-foldli neutral*[*symmetric*] **by** *sepref*

lemma [*sepref-opt-simps*]:
 (*x* = *True*) = *x*
 by *simp*

sepref-definition *dbm-subset′-impl2* **is**
  uncurry2 (*RETURN ooo dbm-subset′*) ::
  [λ((*i*, -), -). *i*≤*n*]$_a$ *nat-assn*^{k} $*_a$ *mtx-assn*^{k} $*_a$ *mtx-assn*^{k} → *bool-assn*
unfolding *dbm-subset′-alt-def list-all-foldli* **by** *sepref*

definition
  dbm-subset′-impl′ ≡ λ*m a b*.
    do {
    imp-for 0 ((*m* + *1*) ∗ (*m* + *1*)) *Heap-Monad.return*
      (λ*i* -. *do* {
        x ← *Array.nth a i*; *y* ← *Array.nth b i*; *Heap-Monad.return* (*x* ≤ *y*)
      })
      True
    }

lemma *imp-for-list-all2-spec*:

  <*a* $\mapsto_a$ *xs* ∗ *b* $\mapsto_a$ *ys*>
  imp-for 0 n′ Heap-Monad.return
    (λ*i* -. *do* {
      x ← *Array.nth a i*; *y* ← *Array.nth b i*; *Heap-Monad.return* (*P x y*)
    })
    True
  <λ*r*. ↑(*r* ⟷ *list-all2 P* (*take n′ xs*) (*take n′ ys*)) ∗ *a* $\mapsto_a$ *xs* ∗ *b* $\mapsto_a$ *ys*>$_t$
  if *n′* ≤ *length xs n′* ≤ *length ys*
  apply (*rule cons-rule*[*rotated 2*])
     apply (*rule imp-for-list-all2′*[**where** *xs* = *xs* **and** *ys* = *ys* **and** *R* = *id-assn* **and** *S* = *id-assn*])
        apply (*use that* **in** *simp*; *fail*)+
    apply (*sep-auto simp*: *pure-def array-assn-def is-array-def*)+
  done

lemma *dbm-subset′-impl′-refine*:
  (*uncurry2 dbm-subset′-impl′*, *uncurry2* (*RETURN* ∘∘∘ *dbm-subset′*))
∈ [λ((*i*, -), -). *i* = *n*]$_a$ *nat-assn*^{k} $*_a$ *local.mtx-assn*^{k} $*_a$ *local.mtx-assn*^{k} → *bool-assn*
  apply *sepref-to-hoare*
  unfolding *dbm-subset′-impl′-def*
  unfolding *amtx-assn-def hr-comp-def is-amtx-def*
  apply (*sep-auto heap*: *imp-for-list-all2-spec simp only*:)
    apply (*simp*; *intro add-mono mult-mono*; *simp*; *fail*)+
  apply *sep-auto*

  subgoal for *b bi ba bia l la a bb*

unfolding *dbm-subset'-def* **by** (*simp add*: *pointwise-cmp-iff*[**where** *xs* = *l* **and** *ys* = *la*])

subgoal for *b bi ba bia l la a bb*
unfolding *dbm-subset'-def* **by** (*simp add*: *pointwise-cmp-iff*[**where** *xs* = *l* **and** *ys* = *la*])
done

sepref-register *check-diag* ::
nat ⇒ *-* :: {*linordered-cancel-ab-monoid-add*,*heap*} *DBMEntry i-mtx* ⇒ *bool*

sepref-register *dbm-subset'* ::
nat ⇒ *'a* :: {*linordered-cancel-ab-monoid-add*,*heap*} *DBMEntry i-mtx* ⇒ *'a DBMEntry i-mtx* ⇒ *bool*

lemmas [*sepref-fr-rules*] = *dbm-subset'-impl'-refine check-diag-impl'.refine*

sepref-definition *dbm-subset-impl'* **is**
uncurry2 (*RETURN ooo dbm-subset*) ::
$[\lambda((i, -), -).\ i{=}n]_a$ *nat-assn*^{k} $*_a$ *mtx-assn*^{k} $*_a$ *mtx-assn*^{k} → *bool-assn*
unfolding *dbm-subset-def dbm-subset'-def*[*symmetric*] *short-circuit-conv* **by** *sepref*

context
notes [*id-rules*] = *itypeI*[*of n TYPE* (*nat*)]
and [*sepref-import-param*] = *IdI*[*of n*]
begin

sepref-definition *dbm-subset-impl* **is**
uncurry (*RETURN oo PR-CONST* (*dbm-subset n*)) :: *mtx-assn*^{k} $*_a$ *mtx-assn*^{k} $\rightarrow_a$ *bool-assn*
unfolding *dbm-subset-def dbm-subset'-def*[*symmetric*] *short-circuit-conv PR-CONST-def* **by** *sepref*

sepref-definition *check-diag-impl* **is**
RETURN o PR-CONST (*check-diag n*) :: *mtx-assn*^{k} $\rightarrow_a$ *bool-assn*
unfolding *check-diag-alt-def list-ex-foldli neutral*[*symmetric*] *PR-CONST-def* **by** *sepref*

sepref-definition *dbm-subset'-impl* **is**
uncurry (*RETURN oo PR-CONST* (*dbm-subset' n*)) :: *mtx-assn*^{k} $*_a$ *mtx-assn*^{k} $\rightarrow_a$ *bool-assn*
unfolding *dbm-subset'-alt-def list-all-foldli PR-CONST-def* **by** *sepref*

end

abbreviation
  iarray-assn x y ≡ *pure* (*br IArray* (λ-. *True*)) *y x*

lemma [*sepref-fr-rules*]:
  (*uncurry* (*return oo IArray.sub*), *uncurry* (*RETURN oo op-list-get*))
  ∈ *iarray-assn*^{k} $*_a$ *id-assn*^{k} $\rightarrow_a$ *id-assn*
unfolding *br-def* **by** *sepref-to-hoare sep-auto*

lemmas *extra-defs* = *extra-upd-def upd-line-def upd-line-0-def*

sepref-definition *norm-upd-impl* **is**
  uncurry2 (*RETURN ooo norm-upd*) ::
    [λ((-, *xs*), *i*). *length xs* > *n* ∧ *i*≤*n*]$_a$ *mtx-assn*^{d} $*_a$ *iarray-assn*^{k} $*_a$ *nat-assn*^{k} → *mtx-assn*
  unfolding *norm-upd-def extra-defs zero-clock-def*[*symmetric*] **by** *sepref*

sepref-definition *norm-upd-impl′* **is**
  uncurry2 (*RETURN ooo norm-upd*) ::
   [λ((-, *xs*), *i*). *length xs* > *n* ∧ *i*≤*n*]$_a$ *mtx-assn*^{d} $*_a$ (*list-assn id-assn*)^k $*_a$ *nat-assn*^{k} → *mtx-assn*
  unfolding *norm-upd-def extra-defs zero-clock-def*[*symmetric*] **by** *sepref*

sepref-definition *extra-lu-upd-impl* **is**
  uncurry3 (λ*x*. *RETURN ooo* (*extra-lu-upd x*)) ::
  [λ(((-, *ys*), *xs*), *i*). *length xs* > *n* ∧ *length ys* > *n* ∧ *i*≤*n*]$_a$
    mtx-assn^d $*_a$ *iarray-assn*^{k} $*_a$ *iarray-assn*^{k} $*_a$ *nat-assn*^{k} → *mtx-assn*
  unfolding *extra-lu-upd-def extra-defs zero-clock-def*[*symmetric*] **by** *sepref*

sepref-definition *mtx-line-to-list-impl* **is**
  uncurry (*RETURN oo PR-CONST mtx-line*) ::
  [λ(*m*, -). *m* ≤ *n*]$_a$ *nat-assn*^{k} $*_a$ *mtx-assn*^{k} → *list-assn id-assn*
 unfolding *mtx-line-def HOL-list.fold-custom-empty PR-CONST-def map-rev-fold*
by *sepref*

context
  fixes *m* :: *nat* **assumes** *m* ≤ *n*
  notes [*id-rules*] = *itypeI*[*of m TYPE* (*nat*)]
    and [*sepref-import-param*] = *IdI*[*of m*]
begin

sepref-definition *mtx-line-to-list-impl2* **is**

RETURN o PR-CONST mtx-line m :: *mtx-assn*^{k} $\rightarrow_a$ *list-assn id-assn*
unfolding *mtx-line-def HOL-list.fold-custom-empty PR-CONST-def map-rev-fold*
apply *sepref-dbg-keep*
using ‹$m \leq n$›
apply *sepref-dbg-trans-keep*
apply *sepref-dbg-opt*
apply *sepref-dbg-cons-solve*
apply *sepref-dbg-cons-solve*
apply *sepref-dbg-constraints*
done

end

lemma *IArray-impl*:
(*return o IArray*, *RETURN o id*) $\in$ (*list-assn id-assn*)^k $\rightarrow_a$ *iarray-assn*
by *sepref-to-hoare* (*sep-auto simp*: *br-def list-assn-pure-conv pure-eq-conv*)

definition
mtx-line-to-iarray-impl m M = (*mtx-line-to-list-impl2 m M* ≫= *return o IArray*)

lemmas *mtx-line-to-iarray-impl-ht* =
mtx-line-to-list-impl2.refine[*to-hnr*, *unfolded hn-refine-def hn-ctxt-def*, *simplified*]

lemmas *IArray-ht* = *IArray-impl*[*to-hnr*, *unfolded hn-refine-def hn-ctxt-def*, *simplified*]

lemma *mtx-line-to-iarray-impl-refine*[*sepref-fr-rules*]:
(*uncurry mtx-line-to-iarray-impl*, *uncurry* (*RETURN* ∘∘ *mtx-line*))
$\in$ [λ(*m*, -). $m \leq n$]$_a$ *nat-assn*^{k} $*_a$ *mtx-assn*^{k} $\rightarrow$ *iarray-assn*
unfolding *mtx-line-to-iarray-impl-def hfref-def*
apply *clarsimp*
apply *sepref-to-hoare*
apply (*sep-auto*
heap: *mtx-line-to-iarray-impl-ht IArray-ht simp*: *br-def pure-eq-conv list-assn-pure-conv*)
apply (*simp add*: *pure-def*)
done

sepref-register *mtx-line* :: *nat* ⇒ (*'ef*) *DBMEntry i-mtx* ⇒ *'ef DBMEntry list*

lemma [*sepref-import-param*]: (*dbm-lt* :: - *DBMEntry* ⇒ -, *dbm-lt*) $\in$ *Id* $\rightarrow$ *Id* $\rightarrow$ *Id* **by** *simp*

sepref-definition *extra-lup-upd-impl* **is**
uncurry3 (λx. *RETURN ooo* (*extra-lup-upd x*)) ::
[λ(((-, *ys*), *xs*), *i*). *length xs* > *n* ∧ *length ys* > *n* ∧ *i*≤*n*]$_a$
mtx-assn^d $*_a$ *iarray-assn*^{k} $*_a$ *iarray-assn*^{k} $*_a$ *nat-assn*^{k} → *mtx-assn*
unfolding *extra-lup-upd-alt-def2 extra-defs zero-clock-def*[*symmetric*] *mtx-line-def*[*symmetric*]
by *sepref*

context
notes [*id-rules*] = *itypeI*[*of n TYPE* (*nat*)]
and [*sepref-import-param*] = *IdI*[*of n*]
begin

definition
unbounded-dbm′ = *unbounded-dbm n*

lemma *unbounded-dbm-alt-def*:
unbounded-dbm n = *op-amtx-new* (*Suc n*) (*Suc n*) (*unbounded-dbm′*)
unfolding *unbounded-dbm′-def* **by** *simp*

We need the custom rule here because *unbounded-dbm* is a higher-order constant

lemma [*sepref-fr-rules*]:
(*uncurry0* (*return unbounded-dbm′*), *uncurry0* (*RETURN* (*PR-CONST* (*unbounded-dbm′*))))
∈ *unit-assn*^{k} $\rightarrow_a$ *pure* (*nat-rel* $\times_r$ *nat-rel* → *Id*)
by *sepref-to-hoare sep-auto*

sepref-register *PR-CONST* (*unbounded-dbm n*) :: *nat* × *nat* ⇒ *int DBMEntry* :: *′b DBMEntry i-mtx*
sepref-register *unbounded-dbm′* :: *nat* × *nat* ⇒ - *DBMEntry*

Necessary to solve side conditions of *op-amtx-new*

lemma *unbounded-dbm′-bounded*:
mtx-nonzero unbounded-dbm′ ⊆ {*0..<Suc n*} × {*0..<Suc n*}
unfolding *mtx-nonzero-def unbounded-dbm′-def unbounded-dbm-def neutral* **by** *auto*

We need to pre-process the lemmas due to a failure of *TRADE*

lemma *unbounded-dbm′-bounded-1*:
(*a*, *b*) ∈ *mtx-nonzero unbounded-dbm′* ⟹ *a* < *Suc n*
using *unbounded-dbm′-bounded* **by** *auto*

lemma *unbounded-dbm′-bounded-2*:
  $(a, b) \in$ *mtx-nonzero unbounded-dbm′* $\Longrightarrow b <$ *Suc n*
  using *unbounded-dbm′-bounded* **by** *auto*

lemmas [*sepref-fr-rules*] = *dbm-subset-impl.refine*

sepref-register *PR-CONST* (*dbm-subset n*) :: *′e DBMEntry i-mtx* ⇒ *′e DBMEntry i-mtx* ⇒ *bool*

lemma [*def-pat-rules*]:
  dbm-subset \$ *n* ≡ *PR-CONST* (*dbm-subset n*)
  by *simp*

sepref-definition *unbounded-dbm-impl* **is**
  uncurry0 (*RETURN* (*PR-CONST* (*unbounded-dbm n*))) :: $unit\text{-}assn^k \rightarrow_a$ *mtx-assn*
  supply *unbounded-dbm′-bounded-1*[*simp*] *unbounded-dbm′-bounded-2*[*simp*]
  using *unbounded-dbm′-bounded*
  apply (*subst unbounded-dbm-alt-def*)
  unfolding *PR-CONST-def* **by** *sepref*

DBM to List

definition *dbm-to-list* :: (*nat* × *nat* ⇒ *′a*) ⇒ *′a list* **where**
  dbm-to-list M ≡
  rev \$ *fold* (*λi xs. fold* (*λj xs. M* (*i, j*) # *xs*) [*0..<Suc n*] *xs*) [*0..<Suc n*] []

sepref-definition *dbm-to-list-impl* **is**
  RETURN o PR-CONST dbm-to-list :: $mtx\text{-}assn^k \rightarrow_a$ *list-assn id-assn*
  unfolding *dbm-to-list-def HOL-list.fold-custom-empty PR-CONST-def* **by** *sepref*

5.8 Pretty-Printing

context
  fixes *show-clock* :: *nat* ⇒ *string*
    and *show-num* :: *′a* :: {*linordered-ab-group-add,heap*} ⇒ *string*
begin

definition
  make-string e i j ≡
    if i = j then if e < 0 then Some (″EMPTY″) else None
    else
    if i = 0 then
    case e of

DBMEntry.Le a ⇒ *if a* = *0 then None else Some* (*show-clock j* @ ''
>= '' @ *show-num* (− *a*))
| *DBMEntry.Lt a* ⇒ *Some* (*show-clock j* @ '' > '' @ *show-num* (− *a*))
| - ⇒ *None*
else if j = *0 then*
case e of
DBMEntry.Le a ⇒ *Some* (*show-clock i* @ '' <= '' @ *show-num a*)
| *DBMEntry.Lt a* ⇒ *Some* (*show-clock i* @ '' < '' @ *show-num a*)
| - ⇒ *None*
else
case e of
DBMEntry.Le a ⇒ *Some* (*show-clock i* @ '' − '' @ *show-clock j* @ ''
<= '' @ *show-num a*)
| *DBMEntry.Lt a* ⇒ *Some* (*show-clock i* @ '' − '' @ *show-clock j* @ '' <
'' @ *show-num a*)
| - ⇒ *None*

definition
dbm-list-to-string xs ≡
(*concat o intersperse* '', '' *o rev o snd o snd*) $ *fold* (λ*e* (*i*, *j*, *acc*).
let
v = *make-string e i j*;
j = (*j* + *1*) *mod* (*n* + *1*);
i = (*if j* = *0 then i* + *1 else i*)
in
case v of
None ⇒ (*i*, *j*, *acc*)
| *Some s* ⇒ (*i*, *j*, *s* # *acc*)
) *xs* (*0*, *0*, [])

lemma [*sepref-import-param*]:
(*dbm-list-to-string*, *PR-CONST dbm-list-to-string*) ∈ ⟨*Id*⟩*list-rel* → ⟨*Id*⟩*list-rel*
by *simp*

definition *show-dbm* **where**
show-dbm M ≡ *PR-CONST dbm-list-to-string* (*dbm-to-list M*)

sepref-register *PR-CONST local.dbm-list-to-string*
sepref-register *dbm-to-list* :: *'b i-mtx* ⇒ *'b list*

lemmas [*sepref-fr-rules*] = *dbm-to-list-impl.refine*

sepref-definition *show-dbm-impl* **is**
RETURN o show-dbm :: $mtx\text{-}assn^k \rightarrow_a$ *list-assn id-assn*
unfolding *show-dbm-def* **by** *sepref*

end

end

end

5.9 Generate Code

lemma [*code*]:
dbm-le a b = $(a = b \vee (a \prec b))$
unfolding *dbm-le-def* **by** *auto*

export-code
norm-upd-impl
reset-canonical-upd-impl
up-canonical-upd-impl
dbm-subset-impl
dbm-subset
show-dbm-impl
checking *SML*

export-code
norm-upd-impl
reset-canonical-upd-impl
up-canonical-upd-impl
dbm-subset-impl
dbm-subset
show-dbm-impl
checking *SML-imp*

end
theory *DBM-Examples*
imports
DBM-Operations-Impl-Refine
FW-More
Show.Show-Instances
begin

5.10 Examples

no-notation *Ref.update* (‹- := -› *62*)

Let us represent the zone $y \leq x \wedge x - y \leq 2 \wedge y \geq 1$ as a DBM:

definition *test-dbm* :: *int DBM'* **where**
  test-dbm = ((((λ(*i*, *j*). *Le 0*)((*1*,*2*) := *Le 2*))((*0*, *2*) := *Le* (−*1*)))((*1*, *0*) := ∞))((*2*, *0*) := ∞)

— Pretty-printing
definition *show-test-dbm* **where**
  show-test-dbm M = *String.implode* (
    show-dbm 2
      (λ*i*. *if i* = *1 then* ''*x*'' *else if i* = *2 then* ''*y*'' *else* ''*f*'') *show*
      M
)

— Pretty-printing
value [*code*] *show-test-dbm test-dbm*

— Canonical form
value [*code*] *show-test-dbm* (*FW' test-dbm 2*)

— Projection onto *x* axis
value [*code*] *show-test-dbm* (*reset'-upd* (*FW' test-dbm 2*) *2* [*2*] *0*)
— Note that if *reset'-upd* is not applied to the canonical form, the result is incorrect:
value [*code*] *show-test-dbm* (*reset'-upd test-dbm 2* [*2*] *0*)
— In this case, we already obtained a new canonical form after reset:
value [*code*] *show-test-dbm* (*FW'* (*reset'-upd* (*FW' test-dbm 2*) *2* [*2*] *0*) *2*)
— Note that *FWI* can be used to restore the canonical form without running a full *FW'*.

— Relaxation, a.k.a computing the "future", or "letting time elapse":
value [*code*] *show-test-dbm* (*up-canonical-upd* (*reset'-upd* (*FW' test-dbm 2*) *2* [*2*] *0*) *2*)
— Note that *up-canonical-upd* always preservers canonical form.

— Intersection
value [*code*] *show-test-dbm* (*FW'* (*And-upd 2*
    (*up-canonical-upd* (*reset'-upd* (*FW' test-dbm 2*) *2* [*2*] *0*) *2*)
    ((λ(*i*, *j*). ∞)((*1*, *0*):=*Lt 1*))) *2*)
— Note that *up-canonical-upd* always preservers canonical form.

— Checking if DBM represents the empty zone
value [*code*] *check-diag 2* (*FW′* (*And-upd 2*
(*up-canonical-upd* (*reset′-upd* (*FW′ test-dbm 2*) *2* [*2*] *0*) *2*)
(($\lambda(i, j).\ \infty$)((*1*, *0*):=*Lt 1*))) *2*)

— Instead of $\lambda(i, j).\ \infty$ we could also have been using *unbounded-dbm*.

end

References

[1] G. Behrmann, P. Bouyer, K. G. Larsen, and R. Pelánek. Lower and upper bounds in zone-based abstractions of timed automata. *Int. J. Softw. Tools Technol. Transf.*, 8(3):204–215, 2006.

[2] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, pages 197–212, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[3] S. Wimmer. *Trustworthy Verification of Realtime Systems.* PhD thesis, Technical University of Munich, Germany, 2020.

[4] S. Wimmer and P. Lammich. Verified model checking of timed automata. In *TACAS (1)*, volume 10805 of *Lecture Notes in Computer Science*, pages 61–78. Springer, 2018.