

Hybrid Multi-Lane Spatial Logic

Sven Linker

May 26, 2024

Abstract

We present a semantic embedding of a spatio-temporal multi-modal logic, specifically defined to reason about motorway traffic, into Isabelle/HOL. The semantic model is an abstraction of a motorway, emphasising local spatial properties, and parameterised by the types of sensors deployed in the vehicles. We use the logic to define controller constraints to ensure safety, i.e., the absence of collisions on the motorway. After proving safety with a restrictive definition of sensors, we relax these assumptions and show how to amend the controller constraints to still guarantee safety.

Published in iFM 2017 [4].

Formal verification of autonomous vehicles on motorways is a challenging problem, due to the complex interactions between dynamical behaviours and controller choices of the vehicles. To overcome the complexities of proving safety properties, we proposed to separate the dynamical behaviour from the concrete changes in space [2]. To that end, we defined *Multi-Lane Spatial Logic* (MLSL), which was used to express guards and invariants of controller automata defining a protocol for safe lane-change manoeuvres. Under the assumption that all vehicles adhere to this protocol, we proved that collisions were avoided. Subsequently, we presented an extension of MLSL to reason about changes in space over time, a system of natural deduction, and formally proved a safety theorem [5, 3]. This proof was carried out manually and dependent on strong assumptions about the vehicles' sensors.

We define a semantic embedding of a further extension of MLSL, inspired by Hybrid Logic [1]. Subsequently, we show how the safety theorem can be proved within this embedding. Finally, we alter this formal embedding by relaxing the assumptions on the sensors. We show that the previously proven safety theorem does *not* ensure safety in this case, and how the controller constraints can be strengthened to guarantee safety.

Contents

1 Discrete Intervals based on Natural Numbers

2

1.1	Basic properties of discrete intervals.	3
1.2	Algebraic properties of intersection and union.	8
2	Closed Real-valued Intervals	22
3	Cars	27
4	Traffic Snapshots	28
5	Views on Traffic	39
6	Restrict Claims and Reservations to a View	51
7	Move a View according to Difference between Traffic Snapshots	60
8	Sensors for Cars	62
9	Visible Length of Cars with Perfect Sensors	63
10	Basic HMLSL	86
10.1	Syntax of Basic HMLSL	86
10.2	Theorems about Basic HMLSL	89
11	Perfect Sensors	105
12	HMLSL for Perfect Sensors	109
13	Safety for Cars with Perfect Sensors	113
14	Regular Sensors	119
15	HMLSL for Regular Sensors	123
16	Safety for Cars with Regular Sensors	126

1 Discrete Intervals based on Natural Numbers

We define a type of intervals based on the natural numbers. To that end, we employ standard operators of Isabelle, but in addition prove some structural properties of the intervals. In particular, we show that this type constitutes a meet-semilattice with a bottom element and equality.

Furthermore, we show that this semilattice allows for a constrained join, i.e., the union of two intervals is defined, if either one of them is empty, or they are consecutive. Finally, we define the notion of *chopping* an interval into two consecutive subintervals.

```

theory NatInt
  imports Main
begin

```

A discrete interval is a set of consecutive natural numbers, or the empty set.

```

typedef nat-int = {S . (∃ (m::nat) n . {m..n }=S) }
  by auto
setup-lifting type-definition-nat-int

```

1.1 Basic properties of discrete intervals.

```

locale nat-int
interpretation nat-int-class?: nat-int .

```

```

context nat-int
begin

```

```

lemma un-consec-seq: (m::nat) ≤ n ∧ n+1 ≤ l → {m..n} ∪ {n+1..l} = {m..l}
  by auto

```

```

lemma int-consec-seq: {(m::nat)..n} ∩ {n+1..l} = {}
  by auto

```

```

lemma empty-type: {} ∈ { S . ∃ (m:: nat) n . {m..n}=S }
  by auto

```

```

lemma inter-result: ∀ x ∈ {S . (∃ (m::nat) n . {m..n }=S) }.
  ∀ y ∈ {S . (∃ (m::nat) n . {m..n }=S) }.
  x ∩ y ∈ {S . (∃ (m::nat) n . {m..n }=S)}
  using Int-atLeastAtMost by blast

```

```

lemma union-result: ∀ x ∈ {S . (∃ (m::nat) n . {m..n }=S) }.
  ∀ y ∈ {S . (∃ (m::nat) n . {m..n }=S) }.
  x ≠ {} ∧ y ≠ {} ∧ Max x + 1 = Min y
  → x ∪ y ∈ {S . (∃ (m::nat) n . {m..n }=S) }

```

```

proof (rule ballI)+

```

```

  fix x y

```

```

  assume x ∈ {S . (∃ (m::nat) n . {m..n }=S) }

```

```

  and y ∈ {S . (∃ (m::nat) n . {m..n }=S) }

```

```

  then have x-def:(∃ m n. {m..n} = x)

```

```

  and y-def:(∃ m n. {m..n} = y) by blast+

```

```

  show x ≠ {} ∧ y ≠ {} ∧ Max x + 1 = Min y

```

```

  → x ∪ y ∈ {S . (∃ m n. {m..n} = S) }

```

```

proof

```

```

  assume pre:x ≠ {} ∧ y ≠ {} ∧ Max x + 1 = Min y

```

```

  then have x-int:∃ m n. m ≤ n ∧ {m..n} = x

```

```

  and y-int:(∃ m n. m ≤ n ∧ {m..n} = y)

```

```

  using x-def y-def by force+

```

```

  {

```

```

fix ya yb xa xb
assume y-prop:ya ≤ yb ∧ {ya..yb} = y and x-prop:xa ≤ xb ∧ {xa..xb} = x

then have upper-x:Max x = xb and lower-y: Min y = ya
  by (auto simp: Max-eq-iff Min-eq-iff)
from upper-x and lower-y and pre have upper-eq-lower: xb+1 = ya
  by blast
hence y = {xb+1 .. yb} using y-prop by blast
hence x ∪ y = {xa..yb}
  using un-consec-seq upper-eq-lower x-prop y-prop by blast
then have x ∪ y ∈ {S.(∃ m n. {m..n} = S)}
  by auto
}
then show x ∪ y ∈ {S.(∃ m n. {m..n} = S)}
  using x-int y-int by blast
qed
qed

```

```

lemma union-empty-result1: ∀ i ∈ {S . (∃ (m::nat) n . {m..n }=S) }.
  i ∪ {} ∈ {S . (∃ (m::nat) n . {m..n }=S) }
by blast

```

```

lemma union-empty-result2: ∀ i ∈ {S . (∃ (m::nat) n . {m..n }=S) }.
  {} ∪ i ∈ {S . (∃ (m::nat) n . {m..n }=S) }
by blast

```

```

lemma finite: ∀ i ∈ {S . (∃ (m::nat) n . {m..n }=S) } . (finite i)
by blast

```

```

lemma not-empty-means-seq: ∀ i ∈ {S . (∃ (m::nat) n . {m..n }=S) } . i ≠ {}
  → ( ∃ m n . m ≤ n ∧ {m..n} = i)
using atLeastatMost-empty-iff
by force
end

```

The empty set is the bottom element of the type. The infimum/meet of the semilattice is set intersection. The order is given by the subset relation.

```

instantiation nat-int :: bot
begin
lift-definition bot-nat-int :: nat-int is Set.empty by force
instance by standard
end

```

```

instantiation nat-int :: inf
begin
lift-definition inf-nat-int :: nat-int ⇒ nat-int ⇒ nat-int is Set.inter by force
instance

```

```

proof qed
end

instantiation nat-int :: order-bot
begin
lift-definition less-eq-nat-int :: nat-int  $\Rightarrow$  nat-int  $\Rightarrow$  bool is Set.subset-eq .
lift-definition less-nat-int :: nat-int  $\Rightarrow$  nat-int  $\Rightarrow$  bool is Set.subset .
instance
proof
  fix i j k :: nat-int
  show  $(i < j) = (i \leq j \wedge \neg j \leq i)$ 
    by (simp add: less-eq-nat-int.rep-eq less-le-not-le less-nat-int.rep-eq)
  show  $i \leq i$  by (simp add: less-eq-nat-int.rep-eq)
  show  $i \leq j \Longrightarrow j \leq k \Longrightarrow i \leq k$  by (simp add: less-eq-nat-int.rep-eq)
  show  $i \leq j \Longrightarrow j \leq i \Longrightarrow i = j$ 
    by (simp add: Rep-nat-int-inject less-eq-nat-int.rep-eq)
  show  $\text{bot} \leq i$  using less-eq-nat-int.rep-eq
    using bot-nat-int.rep-eq by blast
qed
end

instantiation nat-int :: semilattice-inf
begin
instance
proof
  fix i j k :: nat-int
  show  $i \leq j \Longrightarrow i \leq k \Longrightarrow i \leq \text{inf } j \ k$ 
    by (simp add: inf-nat-int.rep-eq less-eq-nat-int.rep-eq)
  show  $\text{inf } i \ j \leq i$ 
    by (simp add: inf-nat-int.rep-eq less-eq-nat-int.rep-eq)
  show  $\text{inf } i \ j \leq j$ 
    by (simp add: inf-nat-int.rep-eq less-eq-nat-int.rep-eq)
qed
end

instantiation nat-int:: equal
begin
definition equal-nat-int :: nat-int  $\Rightarrow$  nat-int  $\Rightarrow$  bool
  where equal-nat-int i j  $\equiv i \leq j \wedge j \leq i$ 
instance
proof
  fix i j :: nat-int
  show equal-class.equal i j =  $(i = j)$  using equal-nat-int-def by auto
qed
end

context nat-int
begin

```

abbreviation *subseteq* :: *nat-int* \Rightarrow *nat-int* \Rightarrow *bool* (**infix** \sqsubseteq 30)

where $i \sqsubseteq j == i \leq j$

abbreviation *empty* :: *nat-int* (\emptyset)

where $\emptyset \equiv \text{bot}$

notation *inf* (**infix** \sqcap 70)

The union of two intervals is only defined, if it is also a discrete interval.

definition *union* :: *nat-int* \Rightarrow *nat-int* \Rightarrow *nat-int* (**infix** \sqcup 65)

where $i \sqcup j = \text{Abs-nat-int } (\text{Rep-nat-int } i \cup \text{Rep-nat-int } j)$

Non-empty intervals contain a minimal and maximal element. Two non-empty intervals i and j are consecutive, if the minimum of j is the successor of the maximum of i . Furthermore, the interval i can be chopped into the intervals j and k , if the union of j and k equals i , and if j and k are not-empty, they must be consecutive. Finally, we define the cardinality of discrete intervals by lifting the cardinality of sets.

definition *maximum* :: *nat-int* \Rightarrow *nat*

where *maximum-def*: $i \neq \emptyset \implies \text{maximum}(i) = \text{Max } (\text{Rep-nat-int } i)$

definition *minimum* :: *nat-int* \Rightarrow *nat*

where *minimum-def*: $i \neq \emptyset \implies \text{minimum}(i) = \text{Min } (\text{Rep-nat-int } i)$

definition *consec* :: *nat-int* \Rightarrow *nat-int* \Rightarrow *bool*

where *consec* $i j \equiv (i \neq \emptyset \wedge j \neq \emptyset \wedge (\text{maximum}(i)+1 = \text{minimum } j))$

definition *N-Chop* :: *nat-int* \Rightarrow *nat-int* \Rightarrow *nat-int* \Rightarrow *bool* (*N'-Chop'*(-, -, ') 51)

where *nchop-def* :

$\text{N-Chop}(i, j, k) \equiv (i = j \sqcup k \wedge (j = \emptyset \vee k = \emptyset \vee \text{consec } j k))$

lift-definition *card'* :: *nat-int* \Rightarrow *nat* (|-| 70) **is** *card* .

For convenience, we also lift the membership relation and its negation to discrete intervals.

lift-definition *el* :: *nat* \Rightarrow *nat-int* \Rightarrow *bool* (**infix** \in 50) **is** *Set.member* .

lift-definition *not-in* :: *nat* \Rightarrow *nat-int* \Rightarrow *bool* (**infix** \notin 40) **is** *Set.not-member* .

end

lemmas_[simp] = *nat-int.el.rep-eq* *nat-int.not-in.rep-eq* *nat-int.card'.rep-eq*

context *nat-int*

begin

lemma *in-not-in-iff1* : $n \in i \longleftrightarrow \neg n \notin i$ **by** *simp*

lemma *in-not-in-iff2* : $n \notin i \longleftrightarrow \neg n \in i$ **by** *simp*

lemma *rep-non-empty-means-seq*: $i \neq \emptyset$
 $\longrightarrow (\exists m n. m \leq n \wedge (\{m..n\} = (\text{Rep-nat-int } i)))$
by (*metis Rep-nat-int Rep-nat-int-inject bot-nat-int.rep-eq nat-int.not-empty-means-seq*)

lemma *non-empty-max*: $i \neq \emptyset \longrightarrow (\exists m . \text{maximum}(i) = m)$
by *auto*

lemma *non-empty-min*: $i \neq \emptyset \longrightarrow (\exists m . \text{minimum}(i) = m)$
by *auto*

lemma *minimum-in*: $i \neq \emptyset \longrightarrow \text{minimum } i \in i$
by (*metis Min-in atLeastatMost-empty-iff2 finite-atLeastAtMost minimum-def el.rep-eq rep-non-empty-means-seq*)

lemma *maximum-in*: $i \neq \emptyset \longrightarrow \text{maximum } i \in i$
by (*metis Max-in atLeastatMost-empty-iff2 finite-atLeastAtMost maximum-def el.rep-eq rep-non-empty-means-seq*)

lemma *non-empty-elem-in*: $i \neq \emptyset \longleftrightarrow (\exists n. n \in i)$

proof

assume *assm*: $i \neq \emptyset$

show $\exists n . n \in i$

by (*metis assm Rep-nat-int-inverse all-not-in-conv el.rep-eq bot-nat-int-def*)

next

assume *assm*: $\exists n. n \in i$

show $i \neq \emptyset$

using *Abs-nat-int-inverse assm el.rep-eq bot-nat-int-def* **by** *fastforce*

qed

lemma *leq-nat-non-empty*: $(m::\text{nat}) \leq n \longrightarrow \text{Abs-nat-int}\{m..n\} \neq \emptyset$

proof

assume *assm*: $m \leq n$

then have *non-empty*: $\{m..n\} \neq \{\}$

using *atLeastatMost-empty-iff* **by** *blast*

with *assm* **have** $\{m..n\} \in \{S . (\exists (m::\text{nat}) n . \{m..n\} = S)\}$ **by** *blast*

then show $\text{Abs-nat-int } \{m..n\} \neq \emptyset$

using *Abs-nat-int-inject empty-type non-empty bot-nat-int-def*

by (*simp add: bot-nat-int.abs-eq*)

qed

lemma *leq-max-sup*: $(m::\text{nat}) \leq n \longrightarrow \text{Max } \{m..n\} = n$

by (*auto simp: Max-eq-iff*)

lemma *leq-min-inf*: $(m::\text{nat}) \leq n \longrightarrow \text{Min } \{m..n\} = m$

by (*auto simp: Min-eq-iff*)

lemma *leq-max-sup'*: $(m::\text{nat}) \leq n \longrightarrow \text{maximum}(\text{Abs-nat-int}\{m..n\}) = n$

proof

assume *assm*: $m \leq n$

hence $in\text{-type}:\{m..n\} \in \{S . (\exists (m::nat) n . m \leq n \wedge \{m..n\}=S) \vee S=\{\}\}$
by *blast*
from *assm* **have** $Abs\text{-nat-int}\{m..n\} \neq \emptyset$ **using** *leq-nat-non-empty* **by** *blast*
hence $max:\text{maximum}(Abs\text{-nat-int}\{m..n\}) = Max (Rep\text{-nat-int} (Abs\text{-nat-int} \{m..n\}))$
using *maximum-def* **by** *blast*
from *in-type* **have** $(Rep\text{-nat-int} (Abs\text{-nat-int} \{m..n\})) = \{m..n\}$
using *Abs-nat-int-inverse* **by** *blast*
hence $Max (Rep\text{-nat-int} (Abs\text{-nat-int}\{m..n\})) = Max \{m..n\}$ **by** *simp*
with *max* **have** $simp\text{-max}:\text{maximum}(Abs\text{-nat-int}\{m..n\}) = Max \{m..n\}$ **by** *simp*
from *assm* **have** $Max \{m..n\} = n$ **using** *leq-max-sup* **by** *blast*
with *simp-max* **show** $\text{maximum}(Abs\text{-nat-int}\{m..n\}) = n$ **by** *simp*
qed

lemma $leq\text{-min-inf}':(m::nat) \leq n \longrightarrow \text{minimum}(Abs\text{-nat-int}\{m..n\}) = m$
proof

assume *assm*: $m \leq n$
hence $in\text{-type}:\{m..n\} \in \{S . (\exists (m::nat) n . m \leq n \wedge \{m..n\}=S) \vee S=\{\}\}$
by *blast*
from *assm* **have** $Abs\text{-nat-int}\{m..n\} \neq \emptyset$ **using** *leq-nat-non-empty* **by** *blast*
hence $min:\text{minimum}(Abs\text{-nat-int}\{m..n\}) = Min(Rep\text{-nat-int} (Abs\text{-nat-int} \{m..n\}))$
using *minimum-def* **by** *blast*
from *in-type* **have** $(Rep\text{-nat-int} (Abs\text{-nat-int} \{m..n\})) = \{m..n\}$
using *Abs-nat-int-inverse* **by** *blast*
hence $Min (Rep\text{-nat-int} (Abs\text{-nat-int}\{m..n\})) = Min \{m..n\}$ **by** *simp*
with *min* **have** $simp\text{-min}:\text{minimum}(Abs\text{-nat-int}\{m..n\}) = Min \{m..n\}$ **by** *simp*
from *assm* **have** $Min \{m..n\} = m$ **using** *leq-min-inf* **by** *blast*
with *simp-min* **show** $\text{minimum}(Abs\text{-nat-int}\{m..n\}) = m$ **by** *simp*
qed

lemma $in\text{-refl}:(n::nat) \in Abs\text{-nat-int} \{n\}$

proof –

have $(n::nat) \leq n$ **by** *simp*
hence $\{n\} \in \{S . (\exists (m::nat) n . m \leq n \wedge \{m..n\}=S) \vee S=\{\}\}$ **by** *auto*
then **show** $n \in Abs\text{-nat-int} \{n\}$
by (*simp add: Abs-nat-int-inverse el-def*)
qed

lemma $in\text{-singleton}: m \in Abs\text{-nat-int}\{n\} \longrightarrow m = n$

proof

assume *assm*: $m \in Abs\text{-nat-int}\{n\}$
have $(n::nat) \leq n$ **by** *simp*
hence $\{n\} \in \{S . (\exists (m::nat) n . m \leq n \wedge \{m..n\}=S) \vee S=\{\}\}$ **by** *auto*
with *assm* **show** $m=n$ **by** (*simp add: Abs-nat-int-inverse el-def*)
qed

1.2 Algebraic properties of intersection and union.

lemma $inter\text{-empty1}:(i::nat\text{-int}) \sqcap \emptyset = \emptyset$

using *Rep-nat-int-inject inf-nat-int.rep-eq bot-nat-int.abs-eq bot-nat-int.rep-eq*

by *fastforce*

lemma *inter-empty2*: $\emptyset \sqcap (i :: \text{nat-int}) = \emptyset$
 by (*metis inf-commute nat-int.inter-empty1*)

lemma *un-empty-absorb1*: $i \sqcup \emptyset = i$
 using *Abs-nat-int-inverse Rep-nat-int-inverse union-def empty-type bot-nat-int.rep-eq*
 by *auto*

lemma *un-empty-absorb2*: $\emptyset \sqcup i = i$
 using *Abs-nat-int-inverse Rep-nat-int-inverse union-def empty-type bot-nat-int.rep-eq*
 by *auto*

Most properties of the union of two intervals depends on them being consecutive, to ensure that their union exists.

lemma *consec-un*: $\text{consec } i \ j \wedge n \notin \text{Rep-nat-int}(i) \cup \text{Rep-nat-int } j$
 $\longrightarrow n \notin (i \sqcup j)$

proof

assume *asm*: $\text{consec } i \ j \wedge n \notin \text{Rep-nat-int } i \cup \text{Rep-nat-int } j$

thus $n \notin (i \sqcup j)$

proof –

have *f1*: $\text{Abs-nat-int } (\text{Rep-nat-int } (i \sqcup j))$
 $= \text{Abs-nat-int } (\text{Rep-nat-int } i \cup \text{Rep-nat-int } j)$

using *Rep-nat-int-inverse union-def* by *presburger*

have $i \neq \emptyset \wedge j \neq \emptyset \wedge \text{maximum } i + 1 = \text{minimum } j$

using *asm consec-def* by *auto*

then have $\exists n \ na. \{n..na\} = \text{Rep-nat-int } i \cup \text{Rep-nat-int } j$

by (*metis (no-types) leq-max-sup leq-min-inf maximum-def minimum-def rep-non-empty-means-seq un-consec-seq*)

then show *?thesis*

using *f1 Abs-nat-int-inject Rep-nat-int not-in.rep-eq asm* by *auto*

qed

qed

lemma *un-subset1*: $\text{consec } i \ j \longrightarrow i \sqsubseteq i \sqcup j$

proof

assume *consec* $i \ j$

then have *asm*: $i \neq \emptyset \wedge j \neq \emptyset \wedge \text{maximum } i + 1 = \text{minimum } j$

using *consec-def* by *blast*

have $\text{Rep-nat-int } i \cup \text{Rep-nat-int } j = \{\text{minimum } i.. \text{maximum } j\}$

by (*metis asm nat-int.leq-max-sup nat-int.leq-min-inf nat-int.maximum-def nat-int.minimum-def nat-int.rep-non-empty-means-seq nat-int.un-consec-seq*)

then show $i \sqsubseteq i \sqcup j$ using *Abs-nat-int-inverse Rep-nat-int*

by (*metis (mono-tags, lifting) Un-upper1 less-eq-nat-int.rep-eq mem-Collect-eq nat-int.union-def*)

qed

lemma *un-subset2*: $\text{consec } i \ j \longrightarrow j \sqsubseteq i \sqcup j$

proof

```

assume consec i j
then have assm:i ≠ ∅ ∧ j ≠ ∅ ∧ maximum i+1 = minimum j
  using consec-def by blast
have Rep-nat-int i ∪ Rep-nat-int j = {minimum i.. maximum j}
  by (metis assm nat-int.leq-max-sup nat-int.leq-min-inf nat-int.maximum-def
    nat-int.minimum-def nat-int.rep-non-empty-means-seq nat-int.un-consec-seq)
then show j ⊆ i ∪ j using Abs-nat-int-inverse Rep-nat-int
  by (metis (mono-tags, lifting) Un-upper2 less-eq-nat-int.rep-eq mem-Collect-eq
    nat-int.union-def)

```

qed

```

lemma inter-distr1:consec j k ⟶ i ∩ (j ∪ k) = (i ∩ j) ∪ (i ∩ k)
  unfolding consec-def

```

proof

```

assume assm:j ≠ ∅ ∧ k ≠ ∅ ∧ maximum j +1 = minimum k
then show i ∩ (j ∪ k) = (i ∩ j) ∪ (i ∩ k)
  proof –
    have f1: ∀ n. n = ∅ ∨ maximum n = Max (Rep-nat-int n)
      using nat-int.maximum-def by auto
    have f2: Rep-nat-int j ≠ {}
      using assm nat-int.maximum-in by auto
    have f3: maximum j = Max (Rep-nat-int j)
      using f1 by (meson assm)
    have maximum k ∈ k
      using assm nat-int.maximum-in by blast
    then have Rep-nat-int k ≠ {}
      by fastforce
    then have Rep-nat-int (j ∪ k) = Rep-nat-int j ∪ Rep-nat-int k
      using f3 f2 Abs-nat-int-inverse Rep-nat-int assm nat-int.minimum-def
        nat-int.union-def union-result
      by auto
    then show ?thesis
      by (metis Rep-nat-int-inverse inf-nat-int.rep-eq inf-sup-distrib1 nat-int.union-def)

```

qed

qed

```

lemma inter-distr2:consec j k ⟶ (j ∪ k) ∩ i = (j ∩ i) ∪ (k ∩ i)
  by (simp add: inter-distr1 inf-commute)

```

```

lemma consec-un-not-elem1:consec i j ∧ n ∉ i ∪ j ⟶ n ∉ i
  using un-subset1 less-eq-nat-int.rep-eq not-in.rep-eq by blast

```

```

lemma consec-un-not-elem2:consec i j ∧ n ∉ i ∪ j ⟶ n ∉ j
  using un-subset2 less-eq-nat-int.rep-eq not-in.rep-eq by blast

```

```

lemma consec-un-element1:consec i j ∧ n ∈ i ⟶ n ∈ i ∪ j
  using less-eq-nat-int.rep-eq nat-int.el.rep-eq nat-int.un-subset1 by blast

```

```

lemma consec-un-element2:consec i j ∧ n ∈ j ⟶ n ∈ i ∪ j

```

using *less-eq-nat-int.rep-eq nat-int.el.rep-eq nat-int.un-subset2* **by** *blast*

lemma *consec-lesser*: $\text{consec } i \ j \longrightarrow (\forall n \ m. (n \in i \wedge m \in j \longrightarrow n < m))$

proof (*rule allI|rule impI*)
assume *consec i j*
fix *n and m*
assume *assump*: $n \in i \wedge m \in j$
then have *max*: $n \leq \text{maximum } i$
by (*metis* $\langle \text{consec } i \ j \rangle$ *atLeastAtMost-iff leq-max-sup maximum-def consec-def el.rep-eq rep-non-empty-means-seq*)
from *assump* **have** *min*: $m \geq \text{minimum } j$
by (*metis* *Min-le* $\langle \text{consec } i \ j \rangle$ *finite-atLeastAtMost minimum-def consec-def el.rep-eq rep-non-empty-means-seq*)
from *min* **and** *max* **show** *less*: $n < m$
using *One-nat-def Suc-le-lessD* $\langle \text{consec } i \ j \rangle$ *add.right-neutral add-Suc-right dual-order.trans leD leI consec-def* **by** *auto*

qed

lemma *consec-in-exclusive1*: $\text{consec } i \ j \wedge n \in i \longrightarrow n \notin j$
using *nat-int.consec-lesser nat-int.in-not-in-iff2* **by** *blast*

lemma *consec-in-exclusive2*: $\text{consec } i \ j \wedge n \in j \longrightarrow n \notin i$
using *consec-in-exclusive1 el.rep-eq not-in.rep-eq* **by** *blast*

lemma *consec-un-max*: $\text{consec } i \ j \longrightarrow \text{maximum } j = \text{maximum } (i \sqcup j)$

proof
assume *assm*: *consec i j*
then have $(\forall n \ m. (n \in i \wedge m \in j \longrightarrow n < m))$
using *nat-int.consec-lesser* **by** *blast*
then have $\forall n. (n \in i \longrightarrow n < \text{maximum } j)$
using *assm local.consec-def nat-int.maximum-in* **by** *auto*
then have $\forall n. (n \in i \sqcup j \longrightarrow n \leq \text{maximum } j)$
by (*metis* (*no-types, lifting*) *Rep-nat-int Rep-nat-int-inverse Un-iff assm atLeastAtMost-iff bot-nat-int.rep-eq less-imp-le-nat local.consec-def local.not-empty-means-seq nat-int.consec-un nat-int.el.rep-eq nat-int.in-not-in-iff1 nat-int.leq-max-sup'*)
then show $\text{maximum } j = \text{maximum } (i \sqcup j)$
by (*metis* *Rep-nat-int-inverse assm atLeastAtMost-iff bot.extremum-uniqueI le-antisym local.consec-def nat-int.consec-un-element2 nat-int.el.rep-eq nat-int.leq-max-sup' nat-int.maximum-in nat-int.un-subset2 rep-non-empty-means-seq*)

qed

lemma *consec-un-min*: $\text{consec } i \ j \longrightarrow \text{minimum } i = \text{minimum } (i \sqcup j)$

proof
assume *assm*: *consec i j*
then have $(\forall n \ m. (n \in i \wedge m \in j \longrightarrow n < m))$
using *nat-int.consec-lesser* **by** *blast*
then have $\forall n. (n \in j \longrightarrow n > \text{minimum } i)$
using *assm local.consec-def nat-int.minimum-in* **by** *auto*

then have $1:\forall n. (n \in i \sqcup j \longrightarrow n \geq \text{minimum } i)$
using *Rep-nat-int Rep-nat-int-inverse Un-iff assm atLeastAtMost-iff bot-nat-int.rep-eq less-imp-le-nat local.consec-def local.not-empty-means-seq nat-int.consec-un nat-int.el.rep-eq nat-int.in-not-in-iff1*
by (*metis (no-types, lifting) leq-min-inf local.minimum-def*)
from *assm* **have** $i \sqcup j \neq \emptyset$
by (*metis bot.extremum-uniqueI nat-int.consec-def nat-int.un-subset2*)
then show $\text{minimum } i = \text{minimum } (i \sqcup j)$
by (*metis 1 antisym assm atLeastAtMost-iff leq-min-inf nat-int.consec-def nat-int.consec-un-element1 nat-int.el.rep-eq nat-int.minimum-def nat-int.minimum-in rep-non-empty-means-seq*)
qed

lemma *consec-un-defined*:
 $\text{consec } i \ j \longrightarrow (\text{Rep-nat-int } (i \sqcup j) \in \{S . (\exists (m::\text{nat}) \ n . \{m..n\}=S)\})$
using *Rep-nat-int* **by** *auto*

lemma *consec-un-min-max*:
 $\text{consec } i \ j \longrightarrow \text{Rep-nat-int}(i \sqcup j) = \{\text{minimum } i .. \text{maximum } j\}$
proof
assume *assm:consec i j*
then have $1:\text{minimum } j = \text{maximum } i + 1$
by (*simp add: nat-int.consec-def*)
have $i:\text{Rep-nat-int } i = \{\text{minimum } i .. \text{maximum } i\}$
by (*metis Rep-nat-int-inverse assm nat-int.consec-def nat-int.leq-max-sup' nat-int.leq-min-inf' rep-non-empty-means-seq*)
have $j:\text{Rep-nat-int } j = \{\text{minimum } j .. \text{maximum } j\}$
by (*metis Rep-nat-int-inverse assm nat-int.consec-def nat-int.leq-max-sup' nat-int.leq-min-inf' rep-non-empty-means-seq*)
show $\text{Rep-nat-int}(i \sqcup j) = \{\text{minimum } i .. \text{maximum } j\}$
by (*metis Rep-nat-int-inverse antisym assm bot.extremum i nat-int.consec-un-max nat-int.consec-un-min nat-int.leq-max-sup' nat-int.leq-min-inf' nat-int.un-subset1 rep-non-empty-means-seq*)
qed

lemma *consec-un-equality*:
 $(\text{consec } i \ j \ \wedge \ k \neq \emptyset)$
 $\longrightarrow (\text{minimum } (i \sqcup j) = \text{minimum } (k) \ \wedge \ \text{maximum } (i \sqcup j) = \text{maximum } (k))$
 $\longrightarrow i \sqcup j = k$
proof (*rule impI*)
assume *cons:consec i j \wedge k \neq \emptyset*
assume *endpoints: minimum(i \sqcup j) = minimum(k) \wedge maximum(i \sqcup j) = maximum(k)*
have $\text{Rep-nat-int } (i \sqcup j) = \{\text{minimum}(i \sqcup j) .. \text{maximum}(i \sqcup j)\}$
by (*metis cons leq-max-sup leq-min-inf local.consec-def nat-int.consec-un-element2 nat-int.maximum-def nat-int.minimum-def nat-int.non-empty-lem-in rep-non-empty-means-seq*)
then have $1:\text{Rep-nat-int}(i \sqcup j) = \{\text{minimum}(k) .. \text{maximum}(k)\}$
using *endpoints* **by** *simp*
have $\text{Rep-nat-int}(k) = \{\text{minimum}(k) .. \text{maximum}(k)\}$

by (metis cons leq-max-sup leq-min-inf nat-int.maximum-def nat-int.minimum-def
 rep-non-empty-means-seq)
 then show $i \sqcup j = k$ using 1
 by (metis Rep-nat-int-inverse)
 qed

lemma consec-trans-lesser:
 $consec\ i\ j \wedge consec\ j\ k \longrightarrow (\forall n\ m. (n \in i \wedge m \in k \longrightarrow n < m))$
proof (rule allI|rule impI)+
 assume cons: consec i j \wedge consec j k
 fix n and m
 assume *assump*: $n \in i \wedge m \in k$
 have $\forall k. k \in j \longrightarrow k < m$ using consec-lesser *assump* cons by blast
 hence *m-greater*: $m < m$ using cons maximum-in consec-def by blast
 then show $n < m$
 by (metis *assump* cons consec-def dual-order.strict-trans nat-int.consec-lesser
 nat-int.maximum-in)
 qed

lemma consec-inter-empty: $consec\ i\ j \Longrightarrow i \sqcap j = \emptyset$
proof –
 assume consec i j
 then have $i \neq bot \wedge j \neq bot \wedge maximum\ i + 1 = minimum\ j$
 using consec-def by force
 then show ?thesis
 by (metis (no-types) Rep-nat-int-inverse bot-nat-int-def inf-nat-int.rep-eq int-conseq-seq
 nat-int.leq-max-sup nat-int.leq-min-inf nat-int.maximum-def nat-int.minimum-def
 nat-int.rep-non-empty-means-seq)
 qed

lemma consec-intermediate1: $consec\ j\ k \wedge consec\ i\ (j \sqcup k) \longrightarrow consec\ i\ j$
proof
 assume *assm*: $consec\ j\ k \wedge consec\ i\ (j \sqcup k)$
 hence *min-max-yz*: $maximum\ j + 1 = minimum\ k$ using consec-def by blast
 hence *min-max-xyz*: $maximum\ i + 1 = minimum\ (j \sqcup k)$
 using consec-def *assm* by blast
 have *min-y-yz*: $minimum\ j = minimum\ (j \sqcup k)$
 by (simp add: *assm* nat-int.consec-un-min)
 hence *min-max-xy*: $maximum\ i + 1 = minimum\ j$
 using *min-max-xyz* by simp
 thus *consec-x-y*: $consec\ i\ j$ using *assm* consec-def by auto
 qed

lemma consec-intermediate2: $consec\ i\ j \wedge consec\ (i \sqcup j)\ k \longrightarrow consec\ j\ k$
proof
 assume *assm*: $consec\ i\ j \wedge consec\ (i \sqcup j)\ k$
 hence *min-max-yz*: $maximum\ i + 1 = minimum\ j$ using consec-def by blast
 hence *min-max-xyz*: $maximum\ (i \sqcup j) + 1 = minimum\ (k)$
 using consec-def *assm* by blast

have $min\text{-}y\text{-}yz:maximum\ j = maximum\ (i \sqcup j)$
using $assm\ nat\text{-}int.consec\text{-}un\text{-}max$ **by** $blast$
then have $min\text{-}max\text{-}xy:maximum\ j+1 = minimum\ k$
using $min\text{-}max\text{-}xyz$ **by** $simp$
thus $consec\text{-}x\text{-}y:consec\ j\ k$ **using** $assm\ consec\text{-}def$ **by** $auto$
qed

lemma $un\text{-}assoc: consec\ i\ j \wedge consec\ j\ k \longrightarrow (i \sqcup j) \sqcup k = i \sqcup (j \sqcup k)$
proof

assume $assm:consec\ i\ j \wedge consec\ j\ k$
from $assm$ **have** $3:maximum\ (i \sqcup j) = maximum\ j$
using $nat\text{-}int.consec\text{-}un\text{-}max$ **by** $auto$
from $assm$ **have** $4:minimum\ (j \sqcup k) = minimum\ (j)$
using $nat\text{-}int.consec\text{-}un\text{-}min$ **by** $auto$
have $i \sqcup j = Abs\text{-}nat\text{-}int\{minimum\ i \ ..\ maximum\ j\}$
by $(metis\ Rep\text{-}nat\text{-}int\text{-}inverse\ assm\ nat\text{-}int.consec\text{-}un\text{-}min\text{-}max)$
then have $5:(i \sqcup j) \sqcup k = Abs\text{-}nat\text{-}int\{minimum\ i \ ..\ maximum\ k\}$
by $(metis\ (no\text{-}types,\ opaque\text{-}lifting)\ 3\ Rep\text{-}nat\text{-}int\text{-}inverse\ antisym\ assm\ bot.\ extremum\ nat\text{-}int.consec\text{-}def\ nat\text{-}int.consec\text{-}un\text{-}min\ nat\text{-}int.consec\text{-}un\text{-}min\text{-}max\ nat\text{-}int.un\text{-}subset1)$
have $j \sqcup k = Abs\text{-}nat\text{-}int\{minimum\ j \ ..\ maximum\ k\}$
by $(metis\ Rep\text{-}nat\text{-}int\text{-}inverse\ assm\ nat\text{-}int.consec\text{-}un\text{-}min\text{-}max)$
then have $6:i \sqcup (j \sqcup k) = Abs\text{-}nat\text{-}int\{minimum\ i \ ..\ maximum\ k\}$
by $(metis\ (no\text{-}types,\ opaque\text{-}lifting)\ 4\ Rep\text{-}nat\text{-}int\text{-}inverse\ antisym\ assm\ bot.\ extremum\ nat\text{-}int.consec\text{-}def\ nat\text{-}int.consec\text{-}un\text{-}max\ nat\text{-}int.consec\text{-}un\text{-}min\text{-}max\ nat\text{-}int.un\text{-}subset2)$
from 5 **and** 6 **show** $(i \sqcup j) \sqcup k = i \sqcup (j \sqcup k)$ **by** $simp$
qed

lemma $consec\text{-}assoc1:consec\ j\ k \wedge consec\ i\ (j \sqcup k) \longrightarrow consec\ (i \sqcup j)\ k$
proof

assume $assm:consec\ j\ k \wedge consec\ i\ (j \sqcup k)$
hence $min\text{-}max\text{-}yz:maximum\ j + 1 = minimum\ k$ **using** $consec\text{-}def$ **by** $blast$
hence $min\text{-}max\text{-}xyz:maximum\ i + 1 = minimum\ (j \sqcup k)$
using $consec\text{-}def\ assm$ **by** $blast$
have $min\text{-}y\text{-}yz:minimum\ j = minimum\ (j \sqcup k)$
by $(simp\ add:\ assm\ nat\text{-}int.consec\text{-}un\text{-}min)$
hence $min\text{-}max\text{-}xy:maximum\ i+1 = minimum\ j$ **using** $min\text{-}max\text{-}xyz$ **by** $simp$
hence $consec\text{-}x\text{-}y:consec\ i\ j$ **using** $assm\ \text{-}consec\text{-}def$ **by** $auto$
hence $max\text{-}y\text{-}xy:maximum\ j = maximum\ (i \sqcup j)$ **using** $consec\text{-}lesser\ assm$
by $(simp\ add:\ nat\text{-}int.consec\text{-}un\text{-}max)$
have $none\text{-}empty:i \neq \emptyset \wedge j \neq \emptyset \wedge k \neq \emptyset$ **using** $assm$ **by** $(simp\ add:\ consec\text{-}def)$
hence $un\text{-}non\text{-}empty:i \sqcup j \neq \emptyset$
using $bot.\ extremum\text{-}uniqueI\ consec\text{-}x\text{-}y\ nat\text{-}int.un\text{-}subset2$ **by** $force$
have $max:maximum\ (i \sqcup j) + 1 = minimum\ k$
using $min\text{-}max\text{-}yz\ max\text{-}y\text{-}xy$ **by** $auto$
thus $consec\ (i \sqcup j)\ k$ **using** $max\ un\text{-}non\text{-}empty\ none\text{-}empty\ consec\text{-}def$ **by** $blast$
qed

lemma $consec\text{-}assoc2:consec\ i\ j \wedge consec\ (i \sqcup j)\ k \longrightarrow consec\ i\ (j \sqcup k)$
proof

```

assume assm:consec i j  $\wedge$  consec (i  $\sqcup$  j) k
hence consec-y-z:consec j k using assm consec-def consec-intermediate2
  by blast
hence max-y-xy:maximum j = maximum (i  $\sqcup$  j)
  by (simp add: assm nat-int.consec-un-max)
have min-y-yz:minimum j = minimum (j  $\sqcup$  k)
  by (simp add: consec-y-z nat-int.consec-un-min)
have none-empty:i  $\neq$   $\emptyset$   $\wedge$  j  $\neq$   $\emptyset$   $\wedge$  k  $\neq$   $\emptyset$  using assm by (simp add: consec-def)
then have un-non-empty:j  $\sqcup$  k  $\neq$   $\emptyset$ 
  by (metis bot-nat-int.rep-eq Rep-nat-int-inject consec-y-z less-eq-nat-int.rep-eq
    un-subset1 subset-empty)
have max:maximum (i) +1 = minimum (j  $\sqcup$  k)
  using assm min-y-yz consec-def by auto
thus consec i (j  $\sqcup$  k) using max un-non-empty none-empty consec-def by blast
qed

```

lemma *consec-assoc-mult*:

$$(i2 = \emptyset \vee \text{consec } i1 \ i2) \wedge (i3 = \emptyset \vee \text{consec } i3 \ i4) \wedge (\text{consec } (i1 \sqcup i2) \ (i3 \sqcup i4)) \\ \longrightarrow (i1 \sqcup i2) \sqcup (i3 \sqcup i4) = (i1 \sqcup (i2 \sqcup i3)) \sqcup i4$$

proof

```

assume assm: $(i2 = \emptyset \vee \text{consec } i1 \ i2) \wedge (i3 = \emptyset \vee \text{consec } i3 \ i4)$ 
   $\wedge (\text{consec } (i1 \sqcup i2) \ (i3 \sqcup i4))$ 

```

```

hence  $(i2 = \emptyset \vee \text{consec } i1 \ i2)$  by simp

```

```

thus  $(i1 \sqcup i2) \sqcup (i3 \sqcup i4) = (i1 \sqcup (i2 \sqcup i3)) \sqcup i4$ 

```

proof

```

assume empty2: $i2 = \emptyset$ 

```

```

hence only-l1: $(i1 \sqcup i2) = i1$  using un-empty-absorb1 by simp

```

```

from assm have  $(i3 = \emptyset \vee \text{consec } i3 \ i4)$  by simp

```

```

thus  $(i1 \sqcup i2) \sqcup (i3 \sqcup i4) = (i1 \sqcup (i2 \sqcup i3)) \sqcup i4$ 

```

```

  by (metis Rep-nat-int-inverse assm bot-nat-int.rep-eq empty2 local.union-def
    nat-int.consec-intermediate1 nat-int.un-assoc only-l1 sup-bot.left-neutral)

```

next

```

assume consec12: consec i1 i2

```

```

from assm have  $(i3 = \emptyset \vee \text{consec } i3 \ i4)$  by simp

```

```

thus  $(i1 \sqcup i2) \sqcup (i3 \sqcup i4) = (i1 \sqcup (i2 \sqcup i3)) \sqcup i4$ 

```

proof

```

assume empty3: $i3 = \emptyset$ 

```

```

hence only-l4: $(i3 \sqcup i4) = i4$  using un-empty-absorb2 by simp

```

```

have  $(i1 \sqcup (i2 \sqcup i3)) = i1 \sqcup i2$  using empty3 by (simp add: un-empty-absorb1)

```

```

thus ?thesis by (simp add: only-l4)

```

next

```

assume consec34: consec i3 i4

```

```

have consec12-3:consec (i1  $\sqcup$  i2) i3

```

```

  using assm consec34 consec-intermediate1 by blast

```

```

show ?thesis

```

```

  by (metis consec12 consec12-3 consec34 consec-intermediate2 un-assoc)

```

qed

qed

qed

lemma *card-subset-le*: $i \sqsubseteq i' \longrightarrow |i| \leq |i'|$
by (*metis bot-nat-int.rep-eq card-mono finite.intros(1) finite-atLeastAtMost less-eq-nat-int.rep-eq local.card'.rep-eq rep-non-empty-means-seq*)

lemma *card-subset-less*: $(i::\text{nat-int}) < i' \longrightarrow |i| < |i'|$
by (*metis bot-nat-int.rep-eq finite.intros(1) finite-atLeastAtMost less-nat-int.rep-eq local.card'.rep-eq psubset-card-mono rep-non-empty-means-seq*)

lemma *card-empty-zero*: $|\emptyset| = 0$
using *Abs-nat-int-inverse empty-type card'.rep-eq bot-nat-int.rep-eq* by *auto*

lemma *card-non-empty-geq-one*: $i \neq \emptyset \longleftrightarrow |i| \geq 1$

proof

assume $i \neq \emptyset$

hence *Rep-nat-int* $i \neq \{\}$ by (*metis Rep-nat-int-inverse bot-nat-int.rep-eq*)

hence *card* (*Rep-nat-int* i) > 0

by (*metis* $\langle i \neq \emptyset \rangle$ *card-0-eq finite-atLeastAtMost gr0I rep-non-empty-means-seq*)

thus $|i| \geq 1$ by (*simp add: card'-def*)

next

assume $|i| \geq 1$ thus $i \neq \emptyset$

using *card-empty-zero* by *auto*

qed

lemma *card-min-max*: $i \neq \emptyset \longrightarrow |i| = (\text{maximum } i - \text{minimum } i) + 1$

proof

assume *assm*: $i \neq \emptyset$

then have *Rep-nat-int* $i = \{\text{minimum } i .. \text{maximum } i\}$

by (*metis leq-max-sup leq-min-inf nat-int.maximum-def nat-int.minimum-def rep-non-empty-means-seq*)

then have *card* (*Rep-nat-int* i) = *maximum* $i - \text{minimum } i + 1$

using *Rep-nat-int-inject assm bot-nat-int.rep-eq* by *fastforce*

then show $|i| = (\text{maximum } i - \text{minimum } i) + 1$ by *simp*

qed

lemma *card-un-add*: *consec* $i j \longrightarrow |i \sqcup j| = |i| + |j|$

proof

assume *assm*: *consec* $i j$

then have $0 : i \sqcap j = \emptyset$

using *nat-int.consec-inter-empty* by *auto*

then have $(\text{Rep-nat-int } i) \cap (\text{Rep-nat-int } j) = \{\}$

by (*metis bot-nat-int.rep-eq inf-nat-int.rep-eq*)

then have 1:

card((*Rep-nat-int* i) \sqcup (*Rep-nat-int* j))=*card*(*Rep-nat-int* i)+*card*(*Rep-nat-int* j)

by (*metis Int-iff add commute add.left-neutral assm card.infinite card-Un-disjoint emptyE le-add1 le-antisym local.consec-def nat-int.card'.rep-eq*)

nat-int.card-min-max nat-int.el.rep-eq nat-int.maximum-in nat-int.minimum-in)

then show $|i \sqcup j| = |i| + |j|$


```

proof –
  have  $f1: i \neq \emptyset \wedge j \neq \emptyset \wedge \text{maximum } i + 1 = \text{minimum } j$ 
    using assm nat-int.consec-def by blast
  then have  $f2: \text{Rep-nat-int } i \neq \{\}$ 
    using Rep-nat-int-inject bot-nat-int.rep-eq by auto
  have  $\text{Rep-nat-int } j \neq \{\}$ 
    using  $f1$  Rep-nat-int-inject bot-nat-int.rep-eq by auto
  then show ?thesis
    using  $f2$   $f1$  Abs-nat-int-inverse Rep-nat-int 1 local.union-result
      nat-int.union-def nat-int-class.maximum-def nat-int-class.minimum-def
    by force
qed
qed

lemma singleton:  $|i| = 1 \longrightarrow (\exists n. \text{Rep-nat-int } i = \{n\})$ 
  using card-1-singletonE card'.rep-eq by fastforce

lemma singleton2:  $(\exists n. \text{Rep-nat-int } i = \{n\}) \longrightarrow |i| = 1$ 
  using card-1-singletonE card'.rep-eq by fastforce

lemma card-seq:
   $\forall i. |i| = x \longrightarrow (\text{Rep-nat-int } i = \{\} \vee (\exists n. \text{Rep-nat-int } i = \{n..n+(x-1)\}))$ 
proof (induct x)
  show IB:
     $\forall i. |i| = 0 \longrightarrow (\text{Rep-nat-int } i = \{\} \vee (\exists n. \text{Rep-nat-int } i = \{n..n+(0-1)\}))$ 
    by (metis card-non-empty-geq-one bot-nat-int.rep-eq not-one-le-zero)
  fix  $x$ 
  assume IH:
     $\forall i. |i| = x \longrightarrow \text{Rep-nat-int } i = \{\} \vee (\exists n. \text{Rep-nat-int } i = \{n..n+(x-1)\})$ 
  show  $\forall i. |i| = \text{Suc } x \longrightarrow$ 
     $\text{Rep-nat-int } i = \{\} \vee (\exists n. \text{Rep-nat-int } i = \{n..n + (\text{Suc } x - 1)\})$ 
proof (rule allI|rule impI)+
  fix  $i$ 
  assume assm-IS:  $|i| = \text{Suc } x$ 
  show  $\text{Rep-nat-int } i = \{\} \vee (\exists n. \text{Rep-nat-int } i = \{n..n + (\text{Suc } x - 1)\})$ 
  proof (cases x = 0)
    assume  $x=0$ 
    hence  $|i| = 1$ 
    using assm-IS by auto
    then have  $\exists n'. \text{Rep-nat-int } i = \{n'\}$ 
    using nat-int.singleton by blast
    hence  $\exists n'. \text{Rep-nat-int } i = \{n'..n' + (\text{Suc } x) - 1\}$ 
    by (simp add: <x = 0>)
    thus  $\text{Rep-nat-int } i = \{\} \vee (\exists n. \text{Rep-nat-int } i = \{n..n + (\text{Suc } x - 1)\})$ 
    by simp
  next
  assume  $x \neq 0: x \neq 0$ 
  hence  $x \geq 0: x > 0$  using gr0I by blast

```

from *assm-IS* **have** $i\text{-is-}\text{seq}:\exists n\ m.\ n \leq m \wedge \text{Rep-nat-int } i = \{n..m\}$
by (*metis One-nat-def Suc-le-mono card-non-empty-geq-one le0 rep-non-empty-means-seq*)
obtain n **and** **where** $\text{seq-def}:\ n \leq m \wedge \text{Rep-nat-int } i = \{n..m\}$
using *i-is-seq* **by** *auto*
have $n\text{-le-}m:n < m$
proof (*rule ccontr*)
assume $\neg n < m$
hence $n = m$ **by** (*simp add: less-le seq-def*)
hence $\text{Rep-nat-int } i = \{n\}$ **by** (*simp add: seq-def*)
hence $x = 0$ **using** *assm-IS card'.rep-eq* **by** *auto*
thus *False* **by** (*simp add: x-neq-0*)
qed
hence $n \leq (m-1)$ **by** *simp*
obtain i' **where** $i\text{-def}:i' = \text{Abs-nat-int } \{n..m-1\}$ **by** *blast*
then **have** $\text{card-}i':|i'| = x$
using *assm-IS leq-nat-non-empty n-le-m*
nat-int-class.card-min-max nat-int-class.leq-max-sup' nat-int-class.leq-min-inf'
seq-def **by** *auto*
hence $\text{Rep-nat-int } i' = \{\}$ $\vee (\exists n.\ \text{Rep-nat-int } i' = \{n.. n + (x - 1)\})$
using *IH* **by** *auto*
hence $(\exists n.\ \text{Rep-nat-int } i' = \{n.. n + (x - 1)\})$ **using** *x-neq-0*
using *card.empty card-i' card'.rep-eq* **by** *auto*
hence $m-1 = n + x - 1$ **using** *assm-IS card'.rep-eq seq-def* **by** *auto*
hence $m = n + x$ **using** *n-le-m x-ge-0* **by** *linarith*
hence $(\text{Rep-nat-int } i = \{n.. n + (\text{Suc } x - 1)\})$ **using** *seq-def* **by** (*simp*)
hence $\exists n.\ (\text{Rep-nat-int } i = \{n.. n + (\text{Suc } x - 1)\})$ **..**
then **show** $\text{Rep-nat-int } i = \{\} \vee (\exists n.\ \text{Rep-nat-int } i = \{n.. n + (\text{Suc } x - 1)\})$
by *blast*
qed
qed
qed

lemma *rep-single*: $\text{Rep-nat-int } (\text{Abs-nat-int } \{m..m\}) = \{m\}$
by (*simp add: Abs-nat-int-inverse*)

lemma *chop-empty-right*: $\forall i.\ N\text{-Chop}(i, i, \emptyset)$
using *bot-nat-int.abs-eq nat-int.inter-empty1 nat-int.nchop-def nat-int.un-empty-absorb1*
by *auto*

lemma *chop-empty-left*: $\forall i.\ N\text{-Chop}(i, \emptyset, i)$
using *bot-nat-int.abs-eq nat-int.inter-empty2 nat-int.nchop-def nat-int.un-empty-absorb2*
by *auto*

lemma *chop-empty* : $N\text{-Chop}(\emptyset, \emptyset, \emptyset)$
by (*simp add: chop-empty-left*)

lemma *chop-always-possible*: $\forall i.\ \exists j\ k.\ N\text{-Chop}(i, j, k)$
by (*metis chop-empty-right*)

lemma chop-add1: $N\text{-Chop}(i,j,k) \longrightarrow |i| = |j| + |k|$
using *card-empty-zero card-un-add un-empty-absorb1 un-empty-absorb2 nchop-def*
by *auto*

lemma chop-add2: $|i| = x+y \longrightarrow (\exists j k. N\text{-Chop}(i,j,k) \wedge |j|=x \wedge |k|=y)$
proof
assume *assm:|i| = x+y*
show $(\exists j k. N\text{-Chop}(i,j,k) \wedge |j|=x \wedge |k|=y)$
proof (*cases x+y = 0*)
assume $x+y = 0$
then show $\exists j k. N\text{-Chop}(i,j,k) \wedge |j|=x \wedge |k|=y$
using *assm chop-empty-left nat-int.chop-add1* **by** *fastforce*
next
assume $x+y \neq 0$
show $\exists j k. N\text{-Chop}(i,j,k) \wedge |j|=x \wedge |k|=y$
proof (*cases x = 0*)
assume $x\text{-eq-0}:x=0$
then show $\exists j k. N\text{-Chop}(i,j,k) \wedge |j|=x \wedge |k|=y$
using *assm nat-int.card-empty-zero nat-int.chop-empty-left* **by** *auto*
next
assume $x\text{-neq-0}:x \neq 0$
show $\exists j k. N\text{-Chop}(i,j,k) \wedge |j|=x \wedge |k|=y$
proof (*cases y = 0*)
assume $y\text{-eq-0}:y=0$
then show $\exists j k. N\text{-Chop}(i,j,k) \wedge |j|=x \wedge |k|=y$
using *assm nat-int.card-empty-zero nat-int.chop-empty-right* **by** *auto*
next
assume $y\text{-neq-0}:y \neq 0$
have *rep-i:* $\exists n. \text{Rep-nat-int } i = \{n..n + (x+y) - 1\}$
using *assm card'.rep-eq card-seq x-neq-0* **by** *fastforce*
obtain *n* **where** *n-def:* $\text{Rep-nat-int } i = \{n..n + (x+y) - 1\}$
using *rep-i* **by** *auto*
have *n-le:* $n \leq n + (x-1)$ **by** *simp*
have *x-le:* $n + (x) \leq n + (x+y) - 1$ **using** *y-neq-0* **by** *linarith*
obtain *j* **where** *j-def:* $j = \text{Abs-nat-int } \{n..n + (x-1)\}$ **by** *blast*
from *n-le* **have** *j-in-type:*
 $\{n..n + (x-1)\} \in \{S . (\exists (m::\text{nat}) n . m \leq n \wedge \{m..n\} = S) \vee S = \{\}\}$
by *blast*
obtain *k* **where** *k-def:* $k = \text{Abs-nat-int } \{n + x..n + (x+y) - 1\}$ **by** *blast*
from *x-le* **have** *k-in-type:*
 $\{n + x..n + (x+y) - 1\} \in \{S . (\exists (m::\text{nat}) n . m \leq n \wedge \{m..n\} = S) \vee S = \{\}\}$
by *blast*
have *consec:* *consec j k*
by (*metis j-def k-def One-nat-def Suc-leI add.assoc diff-add n-le consec-def*
leq-max-sup' leq-min-inf' leq-nat-non-empty neq0-conv x-le x-neq-0)
have *union:* $i = j \sqcup k$
by (*metis Rep-nat-int-inverse consec j-def k-def n-def n-le nat-int.consec-un-min-max*
nat-int.leq-max-sup' nat-int.leq-min-inf' x-le)
have *disj:* $j \cap k = \emptyset$ **using** *consec* **by** (*simp add: consec-inter-empty*)

have chop:N-Chop(i,j,k) **using** consec union disj nchop-def **by** simp
have card-j:|j| = x
using Abs-nat-int-inverse j-def n-le card'.rep-eq x-neq-0 **by** auto
have card-k:|k| = y
using Abs-nat-int-inverse k-def x-le card'.rep-eq x-neq-0 y-neq-0 **by** auto
have N-Chop(i,j,k) \wedge |j| = x \wedge |k| = y **using** chop card-j card-k **by** blast
then show $\exists j k. N\text{-Chop}(i,j,k) \wedge |j|=x \wedge |k|=y$ **by** blast
qed
qed
qed
qed

lemma chop-single:(N-Chop(i,j,k) \wedge |i| = 1) \longrightarrow (|j| = 0 \vee |k| = 0)
using chop-add1 **by** force

lemma chop-leq-max:N-Chop(i,j,k) \wedge consec j k \longrightarrow
 $(\forall n . n \in \text{Rep-nat-int } i \wedge n \leq \text{maximum } j \longrightarrow n \in \text{Rep-nat-int } j)$
by (metis Un-iff le-antisym less-imp-le-nat nat-int.consec-def nat-int.consec-lesser
nat-int.consec-un nat-int.el.rep-eq nat-int.maximum-in nat-int.nchop-def
nat-int.not-in.rep-eq)

lemma chop-geq-min:N-Chop(i,j,k) \wedge consec j k \longrightarrow
 $(\forall n . n \in \text{Rep-nat-int } i \wedge \text{minimum } k \leq n \longrightarrow n \in \text{Rep-nat-int } k)$
by (metis atLeastAtMost-iff bot-nat-int.rep-eq equalsOD leq-max-sup leq-min-inf
nat-int.consec-def nat-int.consec-un-max nat-int.maximum-def nat-int.minimum-def
nat-int.nchop-def rep-non-empty-means-seq)

lemma chop-min:N-Chop(i,j,k) \wedge consec j k \longrightarrow minimum i = minimum j
using nat-int.consec-un-min nat-int.nchop-def **by** auto

lemma chop-max:N-Chop(i,j,k) \wedge consec j k \longrightarrow maximum i = maximum k
using nat-int.consec-un-max nat-int.nchop-def **by** auto

lemma chop-assoc1:
N-Chop($i,i1,i2$) \wedge N-Chop($i2,i3,i4$)
 \longrightarrow (N-Chop($i, i1 \sqcup i3, i4$) \wedge N-Chop($i1 \sqcup i3, i1, i3$))
proof
assume assm:N-Chop($i,i1,i2$) \wedge N-Chop($i2,i3,i4$)
then have chop-def:(i = i1 \sqcup i2 \wedge
(i1 = \emptyset \vee i2 = \emptyset \vee (consec i1 i2)))
using nchop-def **by** blast
hence (i1 = \emptyset \vee i2 = \emptyset \vee (consec i1 i2)) **by** simp
then show N-Chop($i, i1 \sqcup i3, i4$) \wedge N-Chop($i1 \sqcup i3, i1, i3$)
proof
assume empty:i1 = \emptyset
then show N-Chop($i,i1 \sqcup i3, i4$) \wedge N-Chop($i1 \sqcup i3, i1, i3$)
by (simp add: assm chop-def nat-int.chop-empty-left nat-int.un-empty-absorb2)
next
assume i2 = \emptyset \vee (consec i1 i2)

then show $N\text{-Chop}(i, i1 \sqcup i3, i4) \wedge N\text{-Chop}(i1 \sqcup i3, i1, i3)$
proof
 assume $\text{empty}:i2 = \emptyset$
then show $N\text{-Chop}(i, i1 \sqcup i3, i4) \wedge N\text{-Chop}(i1 \sqcup i3, i1, i3)$
by ($\text{metis assm bot.extremum-uniqueI nat-int.chop-empty-right nat-int.nchop-def}$
 $\text{nat-int.un-empty-absorb2 nat-int.un-subset1}$)
next
 assume $\text{consec } i1 \ i2$
then have $\text{consec-}i1\text{-}i2:i1 \neq \emptyset \wedge i2 \neq \emptyset \wedge \text{maximum } i1 + 1 = \text{minimum } i2$
using consec-def **by** blast
from assm **have** $i3 = \emptyset \vee i4 = \emptyset \vee \text{consec } i3 \ i4$
using nchop-def **by** blast
then show $N\text{-Chop}(i, i1 \sqcup i3, i4) \wedge N\text{-Chop}(i1 \sqcup i3, i1, i3)$
proof
 assume $i3\text{-empty}:i3 = \emptyset$
then show $N\text{-Chop}(i, i1 \sqcup i3, i4) \wedge N\text{-Chop}(i1 \sqcup i3, i1, i3)$
using $\text{assm nat-int.chop-empty-right nat-int.nchop-def nat-int.un-empty-absorb2}$
by auto
next
 assume $i4 = \emptyset \vee \text{consec } i3 \ i4$
then show $N\text{-Chop}(i, i1 \sqcup i3, i4) \wedge N\text{-Chop}(i1 \sqcup i3, i1, i3)$
proof
 assume $i4\text{-empty}:i4 = \emptyset$
then show $N\text{-Chop}(i, i1 \sqcup i3, i4) \wedge N\text{-Chop}(i1 \sqcup i3, i1, i3)$
using $\text{assm nat-int.chop-empty-right nat-int.nchop-def}$ **by** auto
next
 assume $\text{consec-}i3\text{-}i4:\text{consec } i3 \ i4$
then show $N\text{-Chop}(i, i1 \sqcup i3, i4) \wedge N\text{-Chop}(i1 \sqcup i3, i1, i3)$
by ($\text{metis } \langle \text{consec } i1 \ i2 \rangle \text{ assm nat-int.consec-}i3\text{-}i4 \text{ nat-int.consec-}i3\text{-}i4 \text{ nat-int.consec-intermediate1}$
 $\text{nat-int.nchop-def nat-int.un-}i3\text{-}i4$)
qed
qed
qed
qed
qed

lemma $\text{chop-}i3\text{-}i4$:

$N\text{-Chop}(i, i1, i2) \wedge N\text{-Chop}(i1, i3, i4)$
 $\longrightarrow N\text{-Chop}(i, i3, i4 \sqcup i2) \wedge N\text{-Chop}(i4 \sqcup i2, i4, i2)$

proof

assume $\text{assm}: N\text{-Chop}(i, i1, i2) \wedge N\text{-Chop}(i1, i3, i4)$

hence ($i1 = \emptyset \vee i2 = \emptyset \vee (\text{consec } i1 \ i2)$)

using nchop-def **by** blast

then show $N\text{-Chop}(i, i3, i4 \sqcup i2) \wedge N\text{-Chop}(i4 \sqcup i2, i4, i2)$

proof

assume $i1\text{-empty}:i1 = \emptyset$

then show $N\text{-Chop}(i, i3, i4 \sqcup i2) \wedge N\text{-Chop}(i4 \sqcup i2, i4, i2)$

by ($\text{metis assm nat-int.chop-empty-left nat-int.consec-un-not-elem1 nat-int.in-not-in-iff1}$
 $\text{nat-int.nchop-def nat-int.non-empty-elem-in nat-int.un-empty-absorb1}$)

```

next
  assume  $i2 = \emptyset \vee \text{consec } i1 \ i2$ 
  then show  $N\text{-Chop}(i, i3, i4 \sqcup i2) \wedge N\text{-Chop}(i4 \sqcup i2, i4, i2)$ 
  proof
    assume  $i2\text{-empty}:i2=\emptyset$ 
    then show  $N\text{-Chop}(i, i3, i4 \sqcup i2) \wedge N\text{-Chop}(i4 \sqcup i2, i4, i2)$ 
      using assm nat-int.chop-empty-right nat-int.nchop-def by auto
  next
    assume  $\text{consec-}i1\text{-}i2:\text{consec } i1 \ i2$ 
    from assm have  $(i3 = \emptyset \vee i4 = \emptyset \vee (\text{consec } i3 \ i4))$ 
      by (simp add: nchop-def)
    then show  $N\text{-Chop}(i, i3, i4 \sqcup i2) \wedge N\text{-Chop}(i4 \sqcup i2, i4, i2)$ 
    proof
      assume  $i3\text{-empty}:i3=\emptyset$ 
      then show  $N\text{-Chop}(i, i3, i4 \sqcup i2) \wedge N\text{-Chop}(i4 \sqcup i2, i4, i2)$ 
        using assm nat-int.chop-empty-left nat-int.nchop-def by auto
    next
      assume  $i4 = \emptyset \vee (\text{consec } i3 \ i4)$ 
      then show  $N\text{-Chop}(i, i3, i4 \sqcup i2) \wedge N\text{-Chop}(i4 \sqcup i2, i4, i2)$ 
      proof
        assume  $i4\text{-empty}:i4=\emptyset$ 
        then show  $N\text{-Chop}(i, i3, i4 \sqcup i2) \wedge N\text{-Chop}(i4 \sqcup i2, i4, i2)$ 
          using assm nat-int.nchop-def nat-int.un-empty-absorb1 nat-int.un-empty-absorb2
            by auto
        next
          assume  $\text{consec-}i3\text{-}i4:\text{consec } i3 \ i4$ 
          then show  $N\text{-Chop}(i, i3, i4 \sqcup i2) \wedge N\text{-Chop}(i4 \sqcup i2, i4, i2)$ 
          by (metis assm consec-}i1\text{-}i2 nat-int.consec-assoc2 nat-int.consec-intermediate2
            nat-int.nchop-def nat-int.un-assoc)
        qed
      qed
    qed
  qed
end
end

lemma chop-subset1:  $N\text{-Chop}(i, j, k) \longrightarrow j \sqsubseteq i$ 
  using nat-int.chop-empty-right nat-int.nchop-def nat-int.un-subset1 by auto

lemma chop-subset2:  $N\text{-Chop}(i, j, k) \longrightarrow k \sqsubseteq i$ 
  using nat-int.chop-empty-left nat-int.nchop-def nat-int.un-subset2 by auto

end
end

```

2 Closed Real-valued Intervals

We define a type for real-valued intervals. It consists of pairs of real numbers, where the first is lesser or equal to the second. Both endpoints are

understood to be part of the interval, i.e., the intervals are closed. This also implies that we do not consider empty intervals.

We define a measure on these intervals as the difference between the left and right endpoint. In addition, we introduce a notion of shifting an interval by a real value x . Finally, an interval r can be chopped into s and t , if the left endpoint of r and s as well as the right endpoint of r and t coincides, and if the right endpoint of s is the left endpoint of t .

```

theory RealInt
  imports HOL.Real
begin

typedef real-int = {r::(real*real) . fst r ≤ snd r}
  by auto
setup-lifting type-definition-real-int

lift-definition left::real-int ⇒ real is fst proof – qed
lift-definition right::real-int ⇒ real is snd proof – qed

lemmas[simp] = left.rep-eq right.rep-eq

locale real-int
interpretation real-int-class?: real-int .

context real-int
begin

definition length :: real-int ⇒ real (||-|| 70)
  where ||r|| ≡ right r – left r

definition shift::real-int ⇒ real ⇒ real-int ( shift - -)
  where (shift r x) = Abs-real-int(left r +x, right r +x)

definition R-Chop :: real-int ⇒ real-int ⇒ real-int ⇒ bool (R'-Chop'(-,-) 51)
  where rchop-def :
    R-Chop(r,s,t) == left r = left s ∧ right s = left t ∧ right r = right t

end

The intervals defined in this way allow for the definition of an order: the
subinterval relation.

instantiation real-int :: order
begin
definition less-eq-real-int r s ≡ (left r ≥ left s) ∧ (right r ≤ right s)
definition less-real-int r s ≡ (left r ≥ left s) ∧ (right r ≤ right s)
  ∧ ¬((left s ≥ left r) ∧ (right s ≤ right r))

instance
proof
  fix r s t :: real-int

```

```

show  $(r < s) = (r \leq s \wedge \neg s \leq r)$  using less-eq-real-int-def less-real-int-def by
auto
show  $r \leq r$  using less-eq-real-int-def by auto
show  $r \leq s \implies s \leq t \implies r \leq t$  using less-eq-real-int-def by auto
show  $r \leq s \implies s \leq r \implies r = s$ 
  by (metis Rep-real-int-inject left.rep-eq less-le less-eq-real-int-def
    not-le prod.collapse right.rep-eq)
qed
end

```

```

context real-int
begin

```

```

lemma left-leq-right:  $\text{left } r \leq \text{right } r$ 
  using Rep-real-int left.rep-eq right.rep-eq by auto

```

```

lemma length-ge-zero :  $\|r\| \geq 0$ 
  using Rep-real-int left.rep-eq right.rep-eq length-def by auto

```

```

lemma consec-add:
   $\text{left } r = \text{left } s \wedge \text{right } r = \text{right } t \wedge \text{right } s = \text{left } t \implies \|r\| = \|s\| + \|t\|$ 
  by (simp add:length-def)

```

```

lemma length-zero-iff-borders-eq:  $\|r\| = 0 \iff \text{left } r = \text{right } r$ 
  using length-def by auto

```

```

lemma shift-left-eq-right:  $\text{left } (\text{shift } r \ x) \leq \text{right } (\text{shift } r \ x)$ 
  using left-leq-right .

```

```

lemma shift-keeps-length:  $\|r\| = \|\text{shift } r \ x\|$ 
  using Abs-real-int-inverse left.rep-eq real-int.length-def length-ge-zero shift-def
  right.rep-eq by auto

```

```

lemma shift-zero:  $(\text{shift } r \ 0) = r$ 
  by (simp add: Rep-real-int-inverse shift-def )

```

```

lemma shift-additivity:  $(\text{shift } r \ (x+y)) = \text{shift } (\text{shift } r \ x) \ y$ 

```

```

proof -

```

```

  have 1:  $(\text{shift } r \ (x+y)) = \text{Abs-real-int } ((\text{left } r) + (x+y), (\text{right } r) + (x+y))$ 
    using shift-def by auto

```

```

  have 2:  $(\text{left } r) + (x+y) \leq (\text{right } r) + (x+y)$  using left-leq-right by auto

```

```

  hence left:  $\text{left } (\text{shift } r \ (x+y)) = (\text{left } r) + (x+y)$ 

```

```

    by (simp add: Abs-real-int-inverse 1)

```

```

  from 2 have right:  $\text{right } (\text{shift } r \ (x+y)) = (\text{right } r) + (x+y)$ 

```

```

    by (simp add: Abs-real-int-inverse 1)

```

```

  have 3:  $(\text{shift } (\text{shift } r \ x) \ y) = \text{Abs-real-int}(\text{left } (\text{shift } r \ x) + y, \text{right}(\text{shift } r \ x) + y)$ 

```

```

    using shift-def by auto

```

```

  have l1:  $\text{left } (\text{shift } r \ x) = \text{left } r + x$ 

```



```

  using shift-def Abs-real-int-inverse 2 fstI mem-Collect-eq prod.sel(2) left.rep-eq
  by auto
  have r1:right (shift r x) = right r + x
  using shift-def Abs-real-int-inverse 2 fstI mem-Collect-eq prod.sel(2) right.rep-eq
  by auto
  from 3 and l1 and r1 have
    (shift (shift r x) y) = Abs-real-int(left r+x+y, right r+x+y)
    by auto
  with 1 show ?thesis by (simp add: add.assoc)
qed

```

lemma chop-always-possible: $\forall r. \exists s t. R\text{-Chop}(r,s,t)$

```

proof
  fix x
  obtain s where l:left x ≤ s ∧ s ≤ right x
    using left-leq-right by auto
  obtain x1 where x1-def:x1 = Abs-real-int(left x,s) by simp
  obtain x2 where x2-def:x2 = Abs-real-int(s, right x) by simp
  have x1-in-type:(left x, s) ∈ {r :: real*real . fst r ≤ snd r } using l by auto
  have x2-in-type:(s, right x) ∈ {r :: real*real . fst r ≤ snd r } using l by auto
  have 1:left x = left x1 using x1-in-type l Abs-real-int-inverse
    by (simp add: x1-def)
  have 2:right x1 = s
    using Abs-real-int-inverse x1-def x1-in-type right.rep-eq by auto
  have 3:right x1 = left x2
    using Abs-real-int-inverse x1-def x1-in-type x2-def x2-in-type left.rep-eq by auto
  from 1 and 2 and 3 have R-Chop(x,x1,x2)
    using Abs-real-int-inverse rchop-def snd-conv x2-def x2-in-type by auto
  then show ∃ x1 x2. R-Chop(x,x1,x2) by blast
qed

```

lemma chop-singleton-right: $\forall r. \exists s. R\text{-Chop}(r,r,s)$

```

proof
  fix x
  obtain y where y = Abs-real-int(right x, right x) by simp
  then have R-Chop(x,x,y)
    by (simp add: Abs-real-int-inverse real-int.rchop-def)
  then show ∃ y. R-Chop(x,x,y) by blast
qed

```

lemma chop-singleton-left: $\forall r. \exists s. R\text{-Chop}(r,s,r)$

```

proof
  fix x
  obtain y where y = Abs-real-int(left x, left x) by simp
  then have R-Chop(x,y,x)
    by (simp add: Abs-real-int-inverse real-int.rchop-def)
  then show ∃ y. R-Chop(x,y,x) by blast
qed

```

lemma chop-add-length: $R\text{-Chop}(r,s,t) \implies \|r\| = \|s\| + \|t\|$
using *consec-add* **by** (*simp add: rchop-def*)

lemma chop-add-length-ge-0: $R\text{-Chop}(r,s,t) \wedge \|s\| > 0 \wedge \|t\| > 0 \implies \|r\| > 0$
using *chop-add-length* **by** *auto*

lemma chop-dense : $\|r\| > 0 \implies (\exists s t. R\text{-Chop}(r,s,t) \wedge \|s\| > 0 \wedge \|t\| > 0)$
proof

assume $\|r\| > 0$
have *ff1*: $\text{left } r < \text{right } r$
using *Rep-real-int* $\langle 0 < \|r\| \rangle$ *length-def* **by** *auto*
have *l-in-type*: $(\text{left } r, \text{right } r) \in \{r :: \text{real} * \text{real} . \text{fst } r \leq \text{snd } r\}$
using *Rep-real-int* **by** *auto*
obtain *x* **where** *x-def*: $x = (\text{left } r + \text{right } r) / 2$
by *blast*
have *x-gr*: $x > \text{left } r$ **using** *ff1* *field-less-half-sum* *x-def* **by** *blast*
have *x-le*: $x < \text{right } r$ **using** *ff1* *x-def* **by** (*simp add: field-sum-of-halves*)
obtain *s* **where** *s-def*: $s = \text{Abs-real-int}(\text{left } r, x)$ **by** *simp*
obtain *t* **where** *t-def*: $t = \text{Abs-real-int}(x, \text{right } r)$ **by** *simp*
have *s-in-type*: $(\text{left } r, x) \in \{r :: \text{real} * \text{real} . \text{fst } r \leq \text{snd } r\}$
using *x-def* *x-le* **by** *auto*
have *t-in-type*: $(x, \text{right } r) \in \{r :: \text{real} * \text{real} . \text{fst } r \leq \text{snd } r\}$
using *x-def* *x-gr* **by** *auto*
have *s-gr-0*: $\|s\| > 0$
using *Abs-real-int-inverse* *s-def* *length-def* *x-gr* **by** *auto*
have *t-gr-0*: $\|t\| > 0$
using *Abs-real-int-inverse* *t-def* *length-def* *x-le* **by** *auto*
have $R\text{-Chop}(r,s,t)$
using *Abs-real-int-inverse* *s-def* *s-in-type* *t-def* *t-in-type* *rchop-def* **by** *auto*
hence $R\text{-Chop}(r,s,t) \wedge \|s\| > 0 \wedge \|t\| > 0$
using *s-gr-0* *t-gr-0* **by** *blast*
thus $\exists s t. R\text{-Chop}(r,s,t) \wedge \|s\| > 0 \wedge \|t\| > 0$ **by** *blast*
qed

lemma chop-assoc1:

$R\text{-Chop}(r,r1,r2) \wedge R\text{-Chop}(r2,r3,r4)$
 $\implies R\text{-Chop}(r, \text{Abs-real-int}(\text{left } r1, \text{right } r3), r4)$
 $\wedge R\text{-Chop}(\text{Abs-real-int}(\text{left } r1, \text{right } r3), r1,r3)$

proof

assume *assm*: $R\text{-Chop}(r,r1,r2) \wedge R\text{-Chop}(r2,r3,r4)$
let *?y1* = $\text{Abs-real-int}(\text{left } r1, \text{right } r3)$
have *l1*: $\text{left } r1 = \text{left } ?y1$
by (*metis* *Abs-real-int-inverse* *assm* *fst-conv* *left.rep-eq* *mem-Collect-eq*
order-trans *real-int.left-leq-right* *real-int.rchop-def* *snd-conv*)
have *r1*: $\text{right } ?y1 = \text{right } r3$
by (*metis* *Rep-real-int-cases* *Rep-real-int-inverse* *assm* *fst-conv* *mem-Collect-eq*
order-trans *real-int.left-leq-right* *real-int.rchop-def* *right.rep-eq* *snd-conv*)
have *g1*: $R\text{-Chop}(r, ?y1, r4)$ **using** *assm* *rchop-def* *r1* *l1* **by** *simp*
have *g2*: $R\text{-Chop}(?y1, r1,r3)$ **using** *assm* *rchop-def* *r1* *l1* **by** *simp*

show $R\text{-Chop}(r, ?y1, r4) \wedge R\text{-Chop}(?y1, r1, r3)$ **using** $g1\ g2$ **by** *simp*
qed

lemma *chop-assoc2*:

$R\text{-Chop}(r, r1, r2) \wedge R\text{-Chop}(r1, r3, r4)$
 $\longrightarrow R\text{-Chop}(r, r3, \text{Abs-real-int}(\text{left } r4, \text{right } r2))$
 $\wedge R\text{-Chop}(\text{Abs-real-int}(\text{left } r4, \text{right } r2), r4, r2)$

proof

assume *assm*: $R\text{-Chop}(r, r1, r2) \wedge R\text{-Chop}(r1, r3, r4)$
let $?y1 = \text{Abs-real-int}(\text{left } r4, \text{right } r2)$
have $\text{left } ?y1 \leq \text{right } ?y1$
using *real-int.left-leq-right* **by** *blast*
have $f1: \text{left } r4 = \text{right } r3$
using *assm real-int.rchop-def* **by** *force*
then have $\text{right}: \text{right } r3 \leq \text{right } r2$
by (*metis (no-types) assm order-trans real-int.left-leq-right real-int.rchop-def*)
then have $l1: \text{left } ?y1 = \text{left } r4$ **using** $f1$ **by** (*simp add: Abs-real-int-inverse*)
have $r1: \text{right } ?y1 = \text{right } r2$
using *Abs-real-int-inverse right f1* **by** *auto*
have $g1: R\text{-Chop}(r, r3, ?y1)$ **using** *assm rchop-def r1 l1* **by** *simp*
have $g2: R\text{-Chop}(?y1, r4, r2)$ **using** *assm rchop-def r1 l1* **by** *simp*
show $R\text{-Chop}(r, r3, ?y1) \wedge R\text{-Chop}(?y1, r4, r2)$ **using** $g1\ g2$ **by** *simp*
qed

lemma *chop-leq1*: $R\text{-Chop}(r, s, t) \longrightarrow s \leq r$
by (*metis (full-types) less-eq-real-int-def order-refl real-int.left-leq-right real-int.rchop-def*)

lemma *chop-leq2*: $R\text{-Chop}(r, s, t) \longrightarrow t \leq r$
by (*metis (full-types) less-eq-real-int-def order-refl real-int.left-leq-right real-int.rchop-def*)

lemma *chop-empty1*: $R\text{-Chop}(r, s, t) \wedge \|s\| = 0 \longrightarrow r = t$
by (*metis (no-types, opaque-lifting) Rep-real-int-inject left.rep-eq prod.collapse real-int.length-zero-iff-borders-eq real-int.rchop-def right.rep-eq*)

lemma *chop-empty2*: $R\text{-Chop}(r, s, t) \wedge \|t\| = 0 \longrightarrow r = s$
by (*metis (no-types, opaque-lifting) Rep-real-int-inject left.rep-eq prod.collapse real-int.length-zero-iff-borders-eq real-int.rchop-def right.rep-eq*)

end

end

3 Cars

We define a type to refer to cars. For simplicity, we assume that (countably) infinite cars exist.

theory *Cars*

```

imports Main
begin

```

The type of cars consists of the natural numbers. However, we do not define or prove any additional structure about it.

```

typedef cars = {n::nat. True} by blast

```

```

locale cars
begin

```

For the construction of possible counterexamples, it is beneficial to prove that at least two cars exist. Furthermore, we show that there indeed exist infinitely many cars.

```

lemma at-least-two-cars-exists: $\exists c d :: cars . c \neq d$ 

```

```

proof –

```

```

  have  $(0::nat) \neq 1$  by simp

```

```

  then have Abs-cars  $(0::nat) \neq Abs-cars(1)$  by (simp add:Abs-cars-inject)

```

```

  thus ?thesis by blast

```

```

qed

```

```

lemma infinite-cars:infinite {c::cars . True}

```

```

proof –

```

```

  have infinite {n::nat. True} by auto

```

```

  then show ?thesis

```

```

    by (metis UNIV-def finite-imageI type-definition.Rep-range type-definition-cars)

```

```

qed

```

```

end

```

```

end

```

4 Traffic Snapshots

Traffic snapshots define the spatial and dynamical arrangement of cars on the whole of the motorway at a single point in time. A traffic snapshot consists of several functions assigning spatial properties and dynamical behaviour to each car. The functions are named as follows.

- *pos*: positions of cars
- *res*: reservations of cars
- *clm*: claims of cars
- *dyn*: current dynamic behaviour of cars
- *physical_size*: the real sizes of cars
- *braking_distance*: braking distance each car needs in emergency

```

theory Traffic
imports NatInt RealInt Cars
begin

```

```

type-synonym lanes = nat-int
type-synonym extension = real-int

```

Definition of the type of traffic snapshots. The constraints on the different functions are the *sanity conditions* of traffic snapshots.

```

typedef traffic =
  { ts :: (cars⇒real)*(cars⇒lanes)*(cars⇒lanes)*(cars⇒real⇒real)*(cars⇒real)*(cars⇒real).
    (∀ c. ((fst (snd ts))) c ⊓ ((fst (snd (snd ts)))) c = ∅ ) ∧
    (∀ c. |(fst (snd ts)) c| ≥ 1) ∧
    (∀ c. |(fst (snd ts)) c| ≤ 2) ∧
    (∀ c. |(fst (snd (snd ts)) c)| ≤ 1) ∧
    (∀ c. |(fst (snd ts)) c| + |(fst (snd (snd ts))) c| ≤ 2) ∧
    (∀ c. (fst(snd(snd (ts)))) c ≠ ∅ →
      (∃ n. Rep-nat-int(fst (snd ts) c) ∪ Rep-nat-int(fst (snd (snd ts)) c)
        = {n, n+1})) ∧
    (∀ c . fst (snd (snd (snd (snd (ts)))))) c > 0) ∧
    (∀ c. snd (snd (snd (snd (snd (ts)))))) c > 0)
  }
proof –
  let ?type =
    { ts :: (cars⇒real)*(cars⇒lanes)*(cars⇒lanes)*(cars⇒real⇒real)*(cars⇒real)*(cars⇒real).
      (∀ c. ((fst (snd ts))) c ⊓ ((fst (snd (snd ts)))) c = ∅ ) ∧
      (∀ c. |(fst (snd ts)) c| ≥ 1) ∧
      (∀ c. |(fst (snd ts)) c| ≤ 2) ∧
      (∀ c. |(fst (snd (snd ts)) c)| ≤ 1) ∧
      (∀ c. |(fst (snd ts)) c| + |(fst (snd (snd ts))) c| ≤ 2) ∧
      (∀ c. (fst(snd(snd (ts)))) c ≠ ∅ →
        (∃ n. Rep-nat-int(fst (snd ts) c) ∪ Rep-nat-int(fst (snd (snd ts)) c)
          = {n, n+1})) ∧
      (∀ c . fst (snd (snd (snd (snd (ts)))))) c > 0) ∧
      (∀ c. snd (snd (snd (snd (snd (ts)))))) c > 0)
    }
  obtain pos where sp-def:∀ c::cars. pos c = (1::real) by force
  obtain re where re-def:∀ c::cars. re c = Abs-nat-int {1} by force
  obtain cl where cl-def:∀ c::cars. cl c = ∅ by force
  obtain dyn where dyn-def:∀ c::cars. ∀ x::real . (dyn c) x = (0::real) by force
  obtain ps where ps-def :∀ c::cars . ps c = (1::real) by force
  obtain sd where sd-def:∀ c::cars . sd c = (1::real) by force
  obtain ts where ts-def:ts = (pos,re,cl, dyn, ps, sd) by simp

  have disj:∀ c .((re c) ⊓ (cl c) = ∅)
    by (simp add: cl-def nat-int.inter-empty1)
  have re-geq-one:∀ c. |re c| ≥ 1
    by (simp add: Abs-nat-int-inverse re-def)
  have re-leq-two:∀ c. |re c| ≤ 2

```

```

using re-def nat-int.rep-single by auto
have cl-leq-one:  $\forall c. |cl\ c| \leq 1$ 
using nat-int.card-empty-zero cl-def by auto
have add-leq-two:  $\forall c. |re\ c| + |cl\ c| \leq 2$ 
using nat-int.card-empty-zero cl-def re-leq-two by (simp)
have consec-re:  $\forall c. |(re\ c)| = 2 \longrightarrow (\exists n. Rep\text{-}nat\text{-}int\ (re\ c) = \{n, n+1\})$ 
by (simp add: Abs-nat-int-inverse re-def)
have clNextRe :
 $\forall c. ((cl\ c) \neq \emptyset \longrightarrow (\exists n. Rep\text{-}nat\text{-}int\ (re\ c) \cup Rep\text{-}nat\text{-}int\ (cl\ c) = \{n, n+1\}))$ 
by (simp add: cl-def)
from dyn-def have dyn-geq-zero:  $\forall c. \forall x. dyn(c)\ x \geq 0$ 
by auto
from ps-def have ps-ge-zero:  $\forall c. ps\ c > 0$  by auto
from sd-def have sd-ge-zero:  $\forall c. sd\ c > 0$  by auto

have ts $\in$ ?type
using sp-def re-def cl-def disj re-geq-one re-leq-two cl-leq-one add-leq-two
consec-re ps-def sd-def ts-def by auto
thus ?thesis by blast
qed

```

```

locale traffic
begin

```

```

notation nat-int.consec (consec)

```

For brevity, we define names for the different functions within a traffic snapshot.

```

definition pos::traffic  $\Rightarrow$  (cars  $\Rightarrow$  real)
where pos ts  $\equiv$  fst (Rep-traffic ts)

```

```

definition res::traffic  $\Rightarrow$  (cars  $\Rightarrow$  lanes)
where res ts  $\equiv$  fst (snd (Rep-traffic ts))

```

```

definition clm ::traffic  $\Rightarrow$  (cars  $\Rightarrow$  lanes)
where clm ts  $\equiv$  fst (snd (snd (Rep-traffic ts)))

```

```

definition dyn::traffic  $\Rightarrow$  (cars  $\Rightarrow$  (real  $\Rightarrow$  real))
where dyn ts  $\equiv$  fst (snd (snd (snd (Rep-traffic ts))))

```

```

definition physical-size::traffic  $\Rightarrow$  (cars  $\Rightarrow$  real)
where physical-size ts  $\equiv$  fst (snd (snd (snd (snd (Rep-traffic ts))))))

```

```

definition braking-distance::traffic  $\Rightarrow$  (cars  $\Rightarrow$  real)
where braking-distance ts  $\equiv$  snd (snd (snd (snd (snd (Rep-traffic ts))))))

```

It is helpful to be able to refer to the sanity conditions of a traffic snapshot via lemmas, hence we prove that the sanity conditions hold for each traffic snapshot.

lemma disjoint: $(res\ ts\ c) \sqcap (clm\ ts\ c) = \emptyset$
using *Rep-traffic res-def clm-def* **by** *auto*

lemma atLeastOneRes: $1 \leq |res\ ts\ c|$
using *Rep-traffic res-def* **by** *auto*

lemma atMostTwoRes: $|res\ ts\ c| \leq 2$
using *Rep-traffic res-def* **by** *auto*

lemma atMostOneCln: $|clm\ ts\ c| \leq 1$
using *Rep-traffic clm-def* **by** *auto*

lemma atMostTwoLanes: $|res\ ts\ c| + |clm\ ts\ c| \leq 2$
using *Rep-traffic res-def clm-def* **by** *auto*

lemma consecutiveRes: $|res\ ts\ c| = 2 \longrightarrow (\exists n . Rep\ nat\ int\ (res\ ts\ c) = \{n, n+1\})$

proof

assume *assump:* $|res\ ts\ c| = 2$

then have *not-empty:* $(res\ ts\ c) \neq \emptyset$

by (*simp add: card-non-empty-geq-one*)

from *assump and card-seq*

have $Rep\ nat\ int\ (res\ ts\ c) = \{\} \vee (\exists n . Rep\ nat\ int\ (res\ ts\ c) = \{n, n+1\})$

by (*metis add-diff-cancel-left' atLeastAtMost-singleton insert-is-Un nat-int.un-consec-seq one-add-one order-refl*)

with *assump show* $(\exists n . Rep\ nat\ int\ (res\ ts\ c) = \{n, n+1\})$

using *Rep-nat-int-inject bot-nat-int.rep-eq card-non-empty-geq-one*

by (*metis not-empty*)

qed

lemma clmNextRes :

$(clm\ ts\ c) \neq \emptyset \longrightarrow (\exists n . Rep\ nat\ int\ (res\ ts\ c) \cup Rep\ nat\ int\ (clm\ ts\ c) = \{n, n+1\})$

using *Rep-traffic res-def clm-def* **by** *auto*

lemma psGeZero: $\forall c . (physical\ size\ ts\ c > 0)$

using *Rep-traffic physical-size-def* **by** *auto*

lemma sdGeZero: $\forall c . (braking\ distance\ ts\ c > 0)$

using *Rep-traffic braking-distance-def* **by** *auto*

While not a sanity condition directly, the following lemma helps to establish general properties of HMLSL later on. It is a consequence of *clmNextRes*.

lemma clm-consec-res:

$(clm\ ts)\ c \neq \emptyset \longrightarrow consec\ (clm\ ts\ c)\ (res\ ts\ c) \vee consec\ (res\ ts\ c)\ (clm\ ts\ c)$

proof

assume *asm:* $clm\ ts\ c \neq \emptyset$

hence *adj:* $(\exists n . Rep\ nat\ int\ (res\ ts\ c) \cup Rep\ nat\ int\ (clm\ ts\ c) = \{n, n+1\})$

using *clmNextRes* **by** *blast*

obtain *n* **where** *n-def:* $Rep\ nat\ int\ (res\ ts\ c) \cup Rep\ nat\ int\ (clm\ ts\ c) = \{n, n+1\}$

using *adj by blast*
have *disj:res ts c \sqcap clm ts c = \emptyset* **using** *disjoint by blast*
from *n-def and disj*
have $(n \in \text{res } ts \ c \wedge n \notin \text{clm } ts \ c) \vee (n \in \text{clm } ts \ c \wedge n \notin \text{res } ts \ c)$
by *(metis UnE bot-nat-int.rep-eq disjoint-insert(1) el.rep-eq inf-nat-int.rep-eq insertI1 insert-absorb not-in.rep-eq)*
thus *consec (clm ts c) (res ts c) \vee consec (res ts c) (clm ts c)*
proof
assume *n-in-res: $n \in \text{res } ts \ c \wedge n \notin \text{clm } ts \ c$*
hence *suc-n-in-clm:n+1 \in clm ts c*
by *(metis UnCI assm el.rep-eq in-not-in-iff1 insert-iff n-def non-empty-elem-in singletonD)*
have *Rep-nat-int (res ts c) \neq {n, n + 1}*
by *(metis assm disj n-def inf-absorb1 inf-commute less-eq-nat-int.rep-eq sup.cobounded2)*
then have *suc-n-not-in-res:n+1 \notin res ts c*
using *n-def n-in-res nat-int.el.rep-eq nat-int.not-in.rep-eq*
by *auto*
from *n-in-res have n-not-in-clm:n \notin clm ts c* **by** *blast*
have *max:nat-int.maximum (res ts c) = n*
using *n-in-res suc-n-not-in-res nat-int.el.rep-eq nat-int.not-in.rep-eq n-def nat-int.maximum-in nat-int.non-empty-elem-in inf-sup-aci(4)*
by *fastforce*
have *min:nat-int.minimum (clm ts c) = n+1*
using *suc-n-in-clm n-not-in-clm nat-int.el.rep-eq nat-int.not-in.rep-eq n-def nat-int.minimum-in nat-int.non-empty-elem-in using inf-sup-aci(4) not-in.rep-eq* **by** *fastforce*
show *?thesis*
using *assm max min n-in-res nat-int.consec-def nat-int.non-empty-elem-in*
by *auto*
next
assume *n-in-clm: $n \in \text{clm } ts \ c \wedge n \notin \text{res } ts \ c$*
have *suc-n-in-res:n+1 \in res ts c*
proof *(rule ccontr)*
assume $\neg n+1 \in \text{res } ts \ c$
then have $n \in \text{res } ts \ c$
by *(metis Int-insert-right-if0 One-nat-def Suc-leI add.right-neutral add-Suc-right atMostTwoRes el.rep-eq inf-bot-right inf-sup-absorb insert-not-empty le-antisym n-def one-add-one order.not-eq-order-implies-strict singleton traffic.atLeastOneRes traffic.consecutiveRes)*
then show *False* **using** *n-in-clm*
using *nat-int.el.rep-eq nat-int.not-in.rep-eq* **by** *auto*
qed
have *max:nat-int.maximum (clm ts c) = n*
by *(metis Rep-nat-int-inverse assm n-in-clm card-non-empty-geq-one*


```

    le-antisym nat-int.in-singleton nat-int.maximum-in singleton traffic.atMostOneClm)
  have min:nat-int.minimum (res ts c) = n+1
  by (metis Int-insert-right-if0 Int-insert-right-if1 Rep-nat-int-inverse
    bot-nat-int.rep-eq el.rep-eq in-not-in-iff1 in-singleton inf-nat-int.rep-eq
    inf-sup-absorb insert-not-empty inter-empty1 minimum-in n-def
    n-in-clm suc-n-in-res)
  then show ?thesis
  using assm max min nat-int.consec-def nat-int.non-empty-elem-in
    suc-n-in-res by auto
qed
qed

```

We define several possible transitions between traffic snapshots. Cars may create or withdraw claims and reservations, as long as the sanity conditions of the traffic snapshots are fulfilled.

In particular, a car can only create a claim, if it possesses only a reservation on a single lane, and does not already possess a claim. Withdrawing a claim can be done in any situation. It only has an effect, if the car possesses a claim. Similarly, the transition for a car to create a reservation is always possible, but only changes the spatial situation on the road, if the car already has a claim. Finally, a car may withdraw its reservation to a single lane, if its current reservation consists of two lanes.

All of these transitions concern the spatial properties of a single car at a time, i.e., for several cars to change their properties, several transitions have to be taken.

definition *create-claim* ::

```

traffic⇒cars⇒nat⇒traffic⇒bool (- -c'(-, -') → - 27)
where (ts -c(c,n)→ ts') == (pos ts') = (pos ts)
    ∧ (res ts') = (res ts)
    ∧ (dyn ts') = (dyn ts)
    ∧ (physical-size ts') = (physical-size ts)
    ∧ (braking-distance ts') = (braking-distance ts)
    ∧ |clm ts c| = 0
    ∧ |res ts c| = 1
    ∧ ((n+1) ∈ res ts c ∨ (n-1) ∈ res ts c)
    ∧ (clm ts') = (clm ts)(c:=Abs-nat-int {n})

```

definition *withdraw-claim* ::

```

traffic⇒cars ⇒traffic⇒bool (- -wdc'(-') → - 27)
where (ts -wdc(c)→ ts') == (pos ts') = (pos ts)
    ∧ (res ts') = (res ts)
    ∧ (dyn ts') = (dyn ts)
    ∧ (physical-size ts') = (physical-size ts)
    ∧ (braking-distance ts') = (braking-distance ts)
    ∧ (clm ts') = (clm ts)(c:=∅)

```

definition *create-reservation* ::

$traffic \Rightarrow cars \Rightarrow traffic \Rightarrow bool (- - r'(-') \rightarrow - 27)$
where $(ts - r(c) \rightarrow ts') == (pos ts') = (pos ts)$
 $\wedge (res ts') = (res ts)(c := ((res ts c) \sqcup (clm ts c)))$
 $\wedge (dyn ts') = (dyn ts)$
 $\wedge (clm ts') = (clm ts)(c := \emptyset)$
 $\wedge (physical-size ts') = (physical-size ts)$
 $\wedge (braking-distance ts') = (braking-distance ts)$

definition *withdraw-reservation* ::

$traffic \Rightarrow cars \Rightarrow nat \Rightarrow traffic \Rightarrow bool (- - wdr'(-, -') \rightarrow - 27)$
where $(ts - wdr(c, n) \rightarrow ts') == (pos ts') = (pos ts)$
 $\wedge (res ts') = (res ts)(c := Abs-nat-int\{n\})$
 $\wedge (dyn ts') = (dyn ts)$
 $\wedge (clm ts') = (clm ts)$
 $\wedge (physical-size ts') = (physical-size ts)$
 $\wedge (braking-distance ts') = (braking-distance ts)$
 $\wedge n \in (res ts c)$
 $\wedge |res ts c| = 2$

The following two transitions concern the dynamical behaviour of the cars. Similar to the spatial properties, a car may change its dynamics, by setting it to a new function f from real to real. Observe that this function is indeed arbitrary and does not constrain the possible behaviour in any way. However, this transition allows a car to change the function determining their braking distance (in fact, all cars are allowed to change this function, if a car changes sets a new dynamical function). That is, our model describes an over-approximation of a concrete situation, where the braking distance is determined by the dynamics.

The final transition describes the passing of x time units. That is, all cars update their position according to their current dynamical behaviour. Observe that this transition requires that the dynamics of each car is at least 0, for each time point between 0 and x . Hence, this condition denotes that all cars drive into the same direction. If the current dynamics of a car violated this constraint, it would have to reset its dynamics, until time may pass again.

definition *change-dyn*::

$traffic \Rightarrow cars \Rightarrow (real \Rightarrow real) \Rightarrow traffic \Rightarrow bool (- - dyn'(-, -') \rightarrow - 27)$
where $(ts - dyn(c, f) \rightarrow ts') == (pos ts' = pos ts)$
 $\wedge (res ts' = res ts)$
 $\wedge (clm ts' = clm ts)$
 $\wedge (dyn ts' = (dyn ts)(c := f))$
 $\wedge (physical-size ts') = (physical-size ts)$

definition *drive*::

$traffic \Rightarrow real \Rightarrow traffic \Rightarrow bool (- - - \rightarrow - 27)$

where $(ts - x \rightarrow ts')$ \equiv $(\forall c. (pos\ ts'\ c = (pos\ ts\ c) + (dyn\ ts\ c\ x)))$
 $\wedge (\forall c\ y. 0 \leq y \wedge y \leq x \longrightarrow dyn\ ts\ c\ y \geq 0)$
 $\wedge (res\ ts' = res\ ts)$
 $\wedge (clm\ ts' = clm\ ts)$
 $\wedge (dyn\ ts' = dyn\ ts)$
 $\wedge (physical\text{-}size\ ts') = (physical\text{-}size\ ts)$
 $\wedge (braking\text{-}distance\ ts') = (braking\text{-}distance\ ts)$

We bundle the dynamical transitions into *evolutions*, since we will only reason about combinations of the dynamical behaviour. This fits to the level of abstraction by hiding the dynamics completely inside of the model.

inductive $evolve::traffic \Rightarrow traffic \Rightarrow bool$ $(- \rightsquigarrow -)$
where $refl : ts \rightsquigarrow ts$ |
 $change: \exists c. \exists f. (ts - dyn(c,f) \rightarrow ts') \Longrightarrow ts' \rightsquigarrow ts'' \Longrightarrow ts \rightsquigarrow ts''$ |
 $drive: \exists x. x \geq 0 \wedge (ts - x \rightarrow ts') \Longrightarrow ts' \rightsquigarrow ts'' \Longrightarrow ts \rightsquigarrow ts''$

lemma $evolve\text{-}trans:(ts0 \rightsquigarrow ts1) \Longrightarrow (ts1 \rightsquigarrow ts2) \Longrightarrow (ts0 \rightsquigarrow ts2)$

proof ($induction\ rule:evolve.induct$)

case $(refl\ ts)$
then show $?case$ **by** $simp$
next
case $(drive\ ts\ ts'\ ts'')$
then show $?case$ **by** $(metis\ evolve.drive)$
next
case $(change\ ts\ ts'\ ts'')$
then show $?case$ **by** $(metis\ evolve.change)$
qed

For general transition sequences, we introduce *abstract transitions*. A traffic snapshot ts' is reachable from ts via an abstract transition, if there is an arbitrary sequence of transitions from ts to ts' .

inductive $abstract::traffic \Rightarrow traffic \Rightarrow bool$ $(- \Rightarrow -)$ **for** ts
where $refl: (ts \Rightarrow ts)$ |
 $evolve: ts \Rightarrow ts' \Longrightarrow ts' \rightsquigarrow ts'' \Longrightarrow ts \Rightarrow ts''$ |
 $cr\text{-}clm: ts \Rightarrow ts' \Longrightarrow \exists c. \exists n. (ts' - c(c,n) \rightarrow ts'') \Longrightarrow ts \Rightarrow ts''$ |
 $wd\text{-}clm: ts \Rightarrow ts' \Longrightarrow \exists c. (ts' - wdc(c) \rightarrow ts'') \Longrightarrow ts \Rightarrow ts''$ |
 $cr\text{-}res: ts \Rightarrow ts' \Longrightarrow \exists c. (ts' - r(c) \rightarrow ts'') \Longrightarrow ts \Rightarrow ts''$ |
 $wd\text{-}res: ts \Rightarrow ts' \Longrightarrow \exists c. \exists n. (ts' - wdr(c,n) \rightarrow ts'') \Longrightarrow ts \Rightarrow ts''$

lemma $abs\text{-}trans: (ts1 \Rightarrow ts2) \Longrightarrow (ts0 \Rightarrow ts1) \Longrightarrow (ts0 \Rightarrow ts2)$

proof ($induction\ rule:abstract.induct$)

case $refl$
then show $?case$ **by** $simp$
next
case $(evolve\ ts'\ ts'')$
then show $?case$
using $traffic.evolve$ **by** $blast$
next

```

    case (cr-clm ts' ts'')
  then show ?case
    using traffic.cr-clm by blast
next
  case (wd-clm ts' ts'')
  then show ?case
    using traffic.wd-clm by blast
next
  case (cr-res ts' ts'')
  then show ?case
    using traffic.cr-res by blast
next
  case (wd-res ts' ts'')
  then show ?case
    using traffic.wd-res by blast
qed

```

Most properties of the transitions are straightforward. However, to show that the transition to create a reservation is always possible, we need to explicitly construct the resulting traffic snapshot. Due to the size of such a snapshot, the proof is lengthy.

lemma *create-res-subseteq1*: $(ts - r(c) \rightarrow ts') \longrightarrow res\ ts\ c \sqsubseteq res\ ts'\ c$

proof

assume *assm*: $(ts - r(c) \rightarrow ts')$

hence $res\ ts'\ c = res\ ts\ c \sqcup clm\ ts\ c$ **using** *create-reservation-def*

using *fun-upd-apply* **by** *auto*

thus $res\ ts\ c \sqsubseteq res\ ts'\ c$

by (*metis* (*no-types*, *lifting*) *Un-commute clm-consec-res nat-int.un-subset2*
nat-int.union-def nat-int.chop-subset1 nat-int.nchop-def)

qed

lemma *create-res-subseteq2*: $(ts - r(c) \rightarrow ts') \longrightarrow clm\ ts\ c \sqsubseteq res\ ts'\ c$

proof

assume *assm*: $(ts - r(c) \rightarrow ts')$

hence $res\ ts'\ c = res\ ts\ c \sqcup clm\ ts\ c$ **using** *create-reservation-def*

using *fun-upd-apply* **by** *auto*

thus $clm\ ts\ c \sqsubseteq res\ ts'\ c$

by (*metis* *Un-commute clm-consec-res disjoint inf-le1 nat-int.un-subset1 nat-int.un-subset2*
nat-int.union-def)

qed

lemma *create-res-subseteq1-neq*: $(ts - r(d) \rightarrow ts') \wedge d \neq c \longrightarrow res\ ts\ c = res\ ts'\ c$

proof

assume *assm*: $(ts - r(d) \rightarrow ts') \wedge d \neq c$

thus $res\ ts\ c = res\ ts'\ c$ **using** *create-reservation-def*

using *fun-upd-apply* **by** *auto*

qed

lemma *create-res-subseteq2-neq*: $(ts - r(d) \rightarrow ts') \wedge d \neq c \longrightarrow clm\ ts\ c = clm\ ts'\ c$

proof

assume $assm:(ts - r(d) \rightarrow ts') \wedge d \neq c$
thus $clm\ ts\ c = clm\ ts'\ c$ **using** *create-reservation-def*
using *fun-upd-apply* **by** *auto*

qed

lemma *always-create-res*: $\forall ts. \exists ts'. (ts - r(c) \rightarrow ts')$

proof

let $?type =$

$\{ts :: (cars \Rightarrow real) * (cars \Rightarrow lanes) * (cars \Rightarrow lanes) * (cars \Rightarrow real \Rightarrow real) * (cars \Rightarrow real) * (cars \Rightarrow real).$
 $(\forall c. ((fst\ (snd\ ts)))\ c \sqcap ((fst\ (snd\ (snd\ ts))))\ c = \emptyset) \wedge$
 $(\forall c. |(fst\ (snd\ ts))\ c| \geq 1) \wedge$
 $(\forall c. |(fst\ (snd\ ts))\ c| \leq 2) \wedge$
 $(\forall c. |(fst\ (snd\ (snd\ ts))\ c)| \leq 1) \wedge$
 $(\forall c. |(fst\ (snd\ ts))\ c| + |(fst\ (snd\ (snd\ ts)))\ c| \leq 2) \wedge$
 $(\forall c. (fst\ (snd\ (snd\ (ts))))\ c \neq \emptyset \rightarrow$
 $(\exists n. Rep\ nat\ int\ (fst\ (snd\ ts)\ c) \cup Rep\ nat\ int\ (fst\ (snd\ (snd\ ts))\ c)$
 $= \{n, n+1\}) \wedge$
 $(\forall c. fst\ (snd\ (snd\ (snd\ (snd\ (ts))))\ c > 0) \wedge$
 $(\forall c. snd\ (snd\ (snd\ (snd\ (snd\ (ts))))\ c > 0)$
 $\}$

fix ts

show $\exists ts'. (ts - r(c) \rightarrow ts')$

proof (*cases* $clm\ ts\ c = \emptyset$)

case *True*

obtain ts' **where** $ts'\text{-def}: ts' = ts$ **by** *simp*

then have $ts - r(c) \rightarrow ts'$

using *create-reservation-def True fun-upd-triv nat-int.un-empty-absorb1*

by *auto*

thus $?thesis ..$

next

case *False*

obtain ts' **where** $ts'\text{-def}: ts' = (pos\ ts,$
 $(res\ ts)(c := (res\ ts\ c) \sqcup (clm\ ts\ c)),$
 $(clm\ ts)(c := \emptyset),$
 $(dyn\ ts), (physical\ size\ ts), (braking\ distance\ ts))$

by *blast*

have $disj: \forall c. (((fst\ (snd\ ts'))\ c) \sqcap ((fst\ (snd\ (snd\ ts'))\ c) = \emptyset)$

by (*simp add: disjoint nat-int.inter-empty1 ts'\text{-def}*)

have $re\text{-geq}\ one: \forall d. |(fst\ (snd\ ts'))\ d| \geq 1$

proof

fix d

show $|fst\ (snd\ ts')\ d| \geq 1$

proof (*cases* $c = d$)

case *True*

then have $fst\ (snd\ ts')\ d = res\ ts\ d \sqcup clm\ ts\ c$

by (*simp add: ts'\text{-def}*)

then have $res\ ts\ d \sqsubseteq fst\ (snd\ ts')\ d$

```

    by (metis False True Un-ac(3) nat-int.un-subset1 nat-int.un-subset2
        nat-int.union-def traffic.clm-consec-res)
  then show ?thesis
  by (metis bot.extremum-uniqueI card-non-empty-geq-one traffic.atLeastOneRes)
next
  case False
  then show ?thesis
    using traffic.atLeastOneRes ts'-def by auto
qed
qed
have re-leq-two:  $\forall c. |(fst (snd ts')) c| \leq 2$ 
  by (metis (no-types, lifting) Un-commute add commute
      atMostTwoLanes atMostTwoRes nat-int.card-un-add clm-consec-res fun-upd-apply
      nat-int.union-def False prod.sel(1) prod.sel(2) ts'-def)
have cl-leq-one:  $\forall c. |(fst (snd (snd ts'))) c| \leq 1$ 
  using atMostOneCln nat-int.card-empty-zero ts'-def by auto
have add-leq-two:  $\forall c. |(fst (snd ts')) c| + |(fst (snd (snd ts'))) c| \leq 2$ 
  by (metis (no-types, lifting) Suc-1 add-Suc add-diff-cancel-left'
      add-mono-thms-linordered-semiring(1) card-non-empty-geq-one cl-leq-one
      fun-upd-apply le-SucE one-add-one prod.sel(1) prod.sel(2) re-leq-two
      traffic.atMostTwoLanes ts'-def)
have clNextRe :
   $\forall c. (((fst (snd (snd ts'))) c) \neq \emptyset \longrightarrow$ 
     $(\exists n. Rep\text{-}nat\text{-}int ((fst (snd ts')) c) \cup Rep\text{-}nat\text{-}int (fst (snd (snd ts')) c)$ 
     $= \{n, n+1\}))$ 
  using clmNextRes ts'-def by auto
have ps-ge-zero:  $(\forall c. fst (snd (snd (snd (snd (ts'))))) c > 0)$ 
  using ts'-def psGeZero by simp
have sd-ge-zero:  $(\forall c. snd (snd (snd (snd (snd (snd (ts'))))) c > 0)$ 
  using ts'-def sdGeZero by simp
have ts'-type:
   $ts' \in ?type$ 
  using ts'-def disj re-geq-one re-leq-two cl-leq-one add-leq-two
      clNextRe mem-Collect-eq ps-ge-zero sd-ge-zero by blast
have rep-eq:  $Rep\text{-}traffic (Abs\text{-}traffic ts') = ts'$ 
  using ts'-def ts'-type Abs-traffic-inverse by blast
have sp-eq:  $(pos (Abs\text{-}traffic ts')) = (pos ts)$ 
  using rep-eq ts'-def Rep-traffic pos-def by auto
have res-eq:  $(res (Abs\text{-}traffic ts')) = (res ts)(c := (res ts c) \sqcup (clm ts c))$ 
  using Rep-traffic ts'-def ts'-type Abs-traffic-inverse rep-eq res-def clm-def
      fstI sndI by auto
have dyn-eq:  $(dyn (Abs\text{-}traffic ts')) = (dyn ts)$ 
  using Rep-traffic ts'-def ts'-type Abs-traffic-inverse rep-eq dyn-def fstI sndI
  by auto
have clm-eq:  $(clm (Abs\text{-}traffic ts')) = (clm ts)(c := \emptyset)$ 
  using ts'-def ts'-type Abs-traffic-inverse rep-eq clm-def fstI sndI Rep-traffic
  by fastforce
then have  $ts \dashv r(c) \rightarrow Abs\text{-}traffic ts'$ 
  using ts'-def ts'-type create-reservation-def

```

*ts'-def disj re-geq-one re-leq-two cl-leq-one add-leq-two
fst-conv snd-conv rep-eq sp-eq res-eq dyn-eq clm-eq
Rep-traffic clm-def res-def clm-def dyn-def physical-size-def braking-distance-def*

by *auto*
then show *?thesis ..*
qed
qed

lemma *create-clm-eq-res:(ts - c(d,n) → ts') → res ts c = res ts' c*
using *create-claim-def by auto*

lemma *withdraw-clm-eq-res:(ts - wdc(d) → ts') → res ts c = res ts' c*
using *withdraw-claim-def by auto*

lemma *withdraw-res-subseteq:(ts - wdr(d,n) → ts') → res ts' c ⊆ res ts c*
using *withdraw-reservation-def order-refl less-eq-nat-int.rep-eq nat-int.el.rep-eq
nat-int.in-refl nat-int.in-singleton fun-upd-apply subset-eq by fastforce*

end
end

5 Views on Traffic

In this section, we define a notion of locality for each car. These local parts of a road are called *views* and define the part of the model currently under consideration by a car. In particular, a view consists of

- the *extension*, a real-valued interval denoting the distance perceived,
- the *lanes*, a discrete interval, denoting which lanes are perceived,
- the *owner*, the car associated with this view.

theory *Views*
imports *NatInt RealInt Cars*
begin

type-synonym *lanes = nat-int*
type-synonym *extension = real-int*

record *view =*
ext::extension
lan ::lanes
own ::cars

The orders on discrete and continuous intervals induce an order on views.
For two views v and v' with $v \leq v'$, we call v a *subview* of v' .

instantiation *view-ext*:: (order) order

begin

definition *less-eq-view-ext* ($V :: 'a \text{ view-ext}$) ($V' :: 'a \text{ view-ext}$) \equiv
 $(\text{ext } V \leq \text{ext } V') \wedge (\text{lan } V \sqsubseteq \text{lan } V') \wedge \text{own } V = \text{own } V'$
 $\wedge \text{more } V \leq \text{more } V'$

definition *less-view-ext* ($V :: 'a \text{ view-ext}$) ($V' :: 'a \text{ view-ext}$) \equiv
 $(\text{ext } V \leq \text{ext } V') \wedge (\text{lan } V \sqsubseteq \text{lan } V') \wedge \text{own } V' = \text{own } V$
 $\wedge \text{more } V \leq \text{more } V' \wedge$
 $\neg((\text{ext } V' \leq \text{ext } V) \wedge (\text{lan } V' \sqsubseteq \text{lan } V) \wedge \text{own } V' = \text{own } V$
 $\wedge \text{more } V' \leq \text{more } V)$

instance

proof

fix $v \ v' \ v'' :: 'a \text{ view-ext}$

show $v \leq v$

using *less-eq-view-ext-def less-eq-nat-int.rep-eq* **by** *auto*

show $(v < v') = (v \leq v' \wedge \neg v' \leq v)$

using *less-eq-view-ext-def less-view-ext-def* **by** *auto*

show $v \leq v' \implies v' \leq v'' \implies v \leq v''$

using *less-eq-view-ext-def less-eq-nat-int.rep-eq order-trans* **by** *auto*

show $v \leq v' \implies v' \leq v \implies v = v'$

using *less-eq-view-ext-def* **by** *auto*

qed

end

locale *view*

begin

notation *nat-int.maximum* (*maximum*)

notation *nat-int.minimum* (*minimum*)

notation *nat-int.consec* (*consec*)

We lift the chopping relations from discrete and continuous intervals to views, and introduce new notation for these relations.

definition $hchop :: \text{view} \Rightarrow \text{view} \Rightarrow \text{view} \Rightarrow \text{bool} \ (-=||-)$

where $(v=u||w) == \text{real-int.R-Chop}(\text{ext } v)(\text{ext } u)(\text{ext } w) \wedge$
 $\text{lan } v = \text{lan } u \wedge$
 $\text{lan } v = \text{lan } w \wedge$
 $\text{own } v = \text{own } u \wedge$
 $\text{own } v = \text{own } w \wedge$
 $\text{more } v = \text{more } w \wedge$
 $\text{more } v = \text{more } u$

definition $vchop :: \text{view} \Rightarrow \text{view} \Rightarrow \text{view} \Rightarrow \text{bool} \ (-=---)$

where $(v=u---w) == \text{nat-int.N-Chop}(\text{lan } v)(\text{lan } u)(\text{lan } w) \wedge$
 $\text{ext } v = \text{ext } u \wedge$
 $\text{ext } v = \text{ext } w \wedge$
 $\text{own } v = \text{own } u \wedge$

$$\begin{aligned}
\text{own } v &= \text{own } w \wedge \\
\text{more } v &= \text{more } w \wedge \\
\text{more } v &= \text{more } u
\end{aligned}$$

We can also switch the perspective of a view to the car c . That is, we substitute c for the original owner of the view.

definition $\text{switch} :: \text{view} \Rightarrow \text{cars} \Rightarrow \text{view} \Rightarrow \text{bool} \ (- = - > -)$
where $(v=c>w) == \text{ext } v = \text{ext } w \wedge$
 $\text{lan } v = \text{lan } w \wedge$
 $\text{own } w = c \wedge$
 $\text{more } v = \text{more } w$

Most of the lemmas in this theory are direct transfers of the corresponding lemmas on discrete and continuous intervals, which implies rather simple proofs. The only exception is the connection between subviews and the chopping operations. This proof is rather lengthy, since we need to distinguish several cases, and within each case, we need to explicitly construct six different views for the chopping relations.

lemma $\text{h-chop-middle1}:(v=u\|w) \longrightarrow \text{left } (\text{ext } v) \leq \text{right } (\text{ext } u)$
by $(\text{metis } \text{hchop-def } \text{real-int.rchop-def } \text{real-int.left-leq-right})$

lemma $\text{h-chop-middle2}:(v=u\|w) \longrightarrow \text{right } (\text{ext } v) \geq \text{left } (\text{ext } w)$
using $\text{real-int.left-leq-right } \text{real-int.rchop-def } \text{view.hchop-def}$ **by** auto

lemma $\text{horizontal-chop1} : \exists u w. (v=u\|w)$

proof –

have $\text{real-chop}:\exists x1 x2. R\text{-Chop}(\text{ext } v, x1,x2)$
using $\text{real-int.chop-singleton-left}$ **by** force
obtain $x1$ **and** $x2$ **where** $x1\text{-}x2\text{-def} : R\text{-Chop}(\text{ext } v, x1,x2)$
using real-chop **by** force
obtain $V1$ **and** $V2$
where $v1:V1 = (\text{ext } = x1, \text{lan} = \text{lan } v, \text{own} = \text{own } v)$
and $v2:V2 = (\text{ext } = x2, \text{lan} = \text{lan } v, \text{own} = \text{own } v)$ **by** blast
from $v1$ **and** $v2$ **have** $v=V1\|V2$
using $\text{hchop-def } x1\text{-}x2\text{-def}$ **by** (simp)
thus $?thesis$ **by** blast

qed

lemma $\text{horizontal-chop-empty-right} :\forall v. \exists u. (v=v\|u)$
using $\text{hchop-def } \text{real-int.chop-singleton-right}$
by $(\text{metis } (\text{no-types}, \text{opaque-lifting}) \text{select-convs})$

lemma $\text{horizontal-chop-empty-left} :\forall v. \exists u. (v=u\|v)$
using $\text{hchop-def } \text{real-int.chop-singleton-left}$
by $(\text{metis } (\text{no-types}, \text{opaque-lifting}) \text{select-convs})$

lemma $\text{horizontal-chop-non-empty}:$

$\|ext\ v\| > 0 \longrightarrow (\exists u\ w. (v=u\|w) \wedge \|ext\ u\| > 0 \wedge \|ext\ w\| > 0)$

proof

assume $\|ext\ v\| > 0$

then obtain $l1$ **and** $l2$

where $chop: R\text{-}Chop(ext\ v, l1, l2) \wedge \|l1\| > 0 \wedge \|l2\| > 0$

using $real\text{-}int.chop\text{-}dense$ **by** $force$

obtain $V1$ **where** $v1\text{-}def: V1 = \langle ext = l1, lan = lan\ v, own = own\ v \rangle$

by $simp$

obtain $V2$ **where** $v2\text{-}def: V2 = \langle ext = l2, lan = lan\ v, own = own\ v \rangle$

by $simp$

then have $(v=V1\|V2) \wedge \|ext\ V1\| > 0 \wedge \|ext\ V2\| > 0$

using $chop\ hchop\text{-}def\ v1\text{-}def$ **by** $(simp)$

then show $(\exists V1\ V2. (v=V1\|V2) \wedge \|ext\ V1\| > 0 \wedge \|ext\ V2\| > 0)$

by $blast$

qed

lemma $horizontal\text{-}chop\text{-}split\text{-}add:$

$x \geq 0 \wedge y \geq 0 \longrightarrow \|ext\ v\| = x+y \longrightarrow (\exists u\ w. (v=u\|w) \wedge \|ext\ u\| = x \wedge \|ext\ w\| = y)$

proof $(rule\ impI)+$

assume $geq\text{-}0: x \geq 0 \wedge y \geq 0$ **and** $len\text{-}v: \|ext\ v\| = x+y$

obtain u

where $v1\text{-}def:$

$u = \langle ext = Abs\text{-}real\text{-}int\ (left\ (ext\ v), left\ (ext\ v) + x), lan = lan\ v, own = (own\ v) \rangle$

by $simp$

have $v1\text{-}in\text{-}type: (left\ (ext\ v), left\ (ext\ v) + x) \in \{r::(real*real) . fst\ r \leq snd\ r\}$

by $(simp\ add: geq\text{-}0)$

obtain w

where $v2\text{-}def:$

$w = \langle ext = Abs\text{-}real\text{-}int\ (left\ (ext\ v)+x, left\ (ext\ v) + (x+y)), lan = (lan\ v), own = (own\ v) \rangle$ **by** $simp$

have $v2\text{-}in\text{-}type:$

$(left\ (ext\ v)+x, left\ (ext\ v) + (x+y)) \in \{r::(real*real) . fst\ r \leq snd\ r\}$

by $(simp\ add: geq\text{-}0)$

from $v1\text{-}def$ **and** $geq\text{-}0$ **have** $len\text{-}v1: \|ext\ u\| = x$ **using** $v1\text{-}in\text{-}type$

by $(simp\ add: Abs\text{-}real\text{-}int\text{-}inverse\ real\text{-}int.length\text{-}def)$

from $v2\text{-}def$ **and** $geq\text{-}0$ **have** $len\text{-}v2: \|ext\ w\| = y$ **using** $v2\text{-}in\text{-}type$

by $(simp\ add: Abs\text{-}real\text{-}int\text{-}inverse\ real\text{-}int.length\text{-}def)$

from $v1\text{-}def$ **and** $v2\text{-}def$ **have** $(v=u\|w)$

using $Abs\text{-}real\text{-}int\text{-}inverse\ fst\text{-}conv\ hchop\text{-}def\ len\text{-}v\ prod.collapse\ real\text{-}int.rchop\text{-}def\ real\text{-}int.length\text{-}def\ snd\text{-}conv\ v1\text{-}in\text{-}type\ v2\text{-}in\text{-}type$ **by** $auto$

with $len\text{-}v1$ **and** $len\text{-}v2$ **have** $(v=u\|w) \wedge \|ext\ u\| = x \wedge \|ext\ w\| = y$ **by** $simp$

thus $(\exists u\ w. (v=u\|w) \wedge \|ext\ u\| = x \wedge \|ext\ w\| = y)$ **by** $blast$

qed

lemma $horizontal\text{-}chop\text{-}assoc1:$

$(v=v1\|v2) \wedge (v2=v3\|v4) \longrightarrow (\exists v'. (v=v'\|v4) \wedge (v'=v1\|v3))$

proof

assume $assm:(v=v1 \parallel v2) \wedge (v2=v3 \parallel v4)$

obtain v'

where v' -def:

$v' = (\text{ext} = \text{Abs-real-int}(\text{left}(\text{ext } v1), \text{right}(\text{ext } v3)),$
 $\text{lan} = (\text{lan } v), \text{own} = (\text{own } v) \text{)}$

by *simp*

hence $1:v=v' \parallel v4$

using *assm real-int.chop-assoc1 hchop-def* **by** *auto*

have $2:v'=v1 \parallel v3$ **using** v' -def *assm real-int.chop-assoc1 hchop-def* **by** *auto*

from 1 **and** 2 **have** $(v=v' \parallel v4) \wedge (v'=v1 \parallel v3)$ **by** *best*

thus $(\exists v'. (v=v' \parallel v4) \wedge (v'=v1 \parallel v3)) \text{ ..}$

qed

lemma *horizontal-chop-assoc2*:

$(v=v1 \parallel v2) \wedge (v1=v3 \parallel v4) \longrightarrow (\exists v'. (v=v3 \parallel v') \wedge (v'=v4 \parallel v2))$

proof

assume $assm:(v=v1 \parallel v2) \wedge (v1=v3 \parallel v4)$

obtain v'

where v' -def:

$v' = (\text{ext} = \text{Abs-real-int}(\text{left}(\text{ext } v4), \text{right}(\text{ext } v2)),$
 $\text{lan} = (\text{lan } v), \text{own} = (\text{own } v) \text{)}$

by *simp*

hence $1:v=v3 \parallel v'$

using *assm fst-conv real-int.chop-assoc2 snd-conv hchop-def* **by** *auto*

have $2: v'=v4 \parallel v2$

using *assm real-int.chop-assoc2 v'-def hchop-def* **by** *auto*

from 1 **and** 2 **have** $(v=v3 \parallel v') \wedge (v'=v4 \parallel v2)$ **by** *best*

thus $(\exists v'. (v=v3 \parallel v') \wedge (v'=v4 \parallel v2)) \text{ ..}$

qed

lemma *horizontal-chop-width-stable*: $(v=u \parallel w) \longrightarrow |\text{lan } v| = |\text{lan } u| \wedge |\text{lan } v| = |\text{lan } w|$

using *hchop-def* **by** *auto*

lemma *horizontal-chop-own-trans*: $(v=u \parallel w) \longrightarrow \text{own } u = \text{own } w$

using *hchop-def* **by** *auto*

lemma *vertical-chop1*: $\forall v. \exists u w. (v=u \text{ -- } w)$

using *vhop-def nat-int.chop-always-possible*

by (*metis (no-types, opaque-lifting) select-convs*)

lemma *vertical-chop-empty-down*: $\forall v. \exists u. (v=v \text{ -- } u)$

using *vhop-def nat-int.chop-empty-right*

by (*metis (no-types, opaque-lifting) select-convs*)

lemma *vertical-chop-empty-up*: $\forall v. \exists u. (v = u \dashv\vdash v)$
using *vchop-def nat-int.chop-empty-left*
by (*metis (no-types, opaque-lifting) select-convs*)

lemma *vertical-chop-assoc1*:
 $(v = v1 \dashv\vdash v2) \wedge (v2 = v3 \dashv\vdash v4) \longrightarrow (\exists v'. (v = v' \dashv\vdash v4) \wedge (v' = v1 \dashv\vdash v3))$

proof
assume *assm*: $(v = v1 \dashv\vdash v2) \wedge (v2 = v3 \dashv\vdash v4)$
obtain v'
where *v'-def*: $v' = (\text{ext} = \text{ext } v, \text{lan} = (\text{lan } v1) \sqcup (\text{lan } v3), \text{own} = (\text{own } v))$
by *simp*
then have $1: v = v' \dashv\vdash v4$
using *assm nat-int.chop-assoc1 vchop-def* **by** *auto*
have $2: v' = v1 \dashv\vdash v3$
using *v'-def assm nat-int.chop-assoc1 vchop-def* **by** *auto*
from 1 **and** 2 **have** $(v = v' \dashv\vdash v4) \wedge (v' = v1 \dashv\vdash v3)$ **by** *best*
then show $(\exists v'. (v = v' \dashv\vdash v4) \wedge (v' = v1 \dashv\vdash v3))$ **..**
qed

lemma *vertical-chop-assoc2*:
 $(v = v1 \dashv\vdash v2) \wedge (v1 = v3 \dashv\vdash v4) \longrightarrow (\exists v'. (v = v3 \dashv\vdash v') \wedge (v' = v4 \dashv\vdash v2))$

proof
assume *assm*: $(v = v1 \dashv\vdash v2) \wedge (v1 = v3 \dashv\vdash v4)$
obtain v'
where *v'-def*: $v' = (\text{ext} = \text{ext } v, \text{lan} = (\text{lan } v4) \sqcup (\text{lan } v2), \text{own} = (\text{own } v))$
by *simp*
then have $1: v = v3 \dashv\vdash v'$
using *assm fst-conv nat-int.chop-assoc2 snd-conv vchop-def* **by** *auto*
have $2: v' = v4 \dashv\vdash v2$
using *assm nat-int.chop-assoc2 v'-def vchop-def* **by** *auto*
from 1 **and** 2 **have** $(v = v3 \dashv\vdash v') \wedge (v' = v4 \dashv\vdash v2)$ **by** *best*
then show $(\exists v'. (v = v3 \dashv\vdash v') \wedge (v' = v4 \dashv\vdash v2))$ **..**
qed

lemma *vertical-chop-singleton*:
 $(v = u \dashv\vdash w) \wedge |\text{lan } v| = 1 \longrightarrow (|\text{lan } u| = 0 \vee |\text{lan } w| = 0)$
using *nat-int.chop-single vchop-def Rep-nat-int-inverse*
by *fastforce*

lemma *vertical-chop-add1*: $(v = u \dashv\vdash w) \longrightarrow |\text{lan } v| = |\text{lan } u| + |\text{lan } w|$
using *nat-int.chop-add1 vchop-def* **by** *fastforce*

lemma *vertical-chop-add2*:
 $|\text{lan } v| = x + y \longrightarrow (\exists u w. (v = u \dashv\vdash w) \wedge |\text{lan } u| = x \wedge |\text{lan } w| = y)$

proof
assume *assm*: $|\text{lan } v| = x + y$
hence *add*: $\exists i j. N\text{-Chop}(\text{lan } v, i, j) \wedge |i| = x \wedge |j| = y$

using *chop-add2* **by** *blast*
obtain *i* **and** *j* **where** $l1-l2-def:N-Chop(lan\ v, i, j) \wedge |i| = x \wedge |j| = y$
using *add* **by** *blast*
obtain *u* **and** *w* **where** $u = (\text{ext} = \text{ext}\ v, lan = i, own = (own\ v))$
and $w = (\text{ext} = \text{ext}\ v, lan = j, own = (own\ v))$ **by** *blast*
hence $(v = u -- w) \wedge |lan\ u| = x \wedge |lan\ w| = y$
using *l1-l2-def view.vchop-def*
by (*simp*)
thus $(\exists\ u\ w. (v = u -- w) \wedge |lan\ u| = x \wedge |lan\ w| = y)$ **by** *blast*
qed

lemma *vertical-chop-length-stable*:
 $(v = u -- w) \longrightarrow \|ext\ v\| = \|ext\ u\| \wedge \|ext\ v\| = \|ext\ w\|$
using *vchop-def* **by** *auto*

lemma *vertical-chop-own-trans*: $(v = u -- w) \longrightarrow own\ u = own\ w$
using *vchop-def* **by** *auto*

lemma *vertical-chop-width-mon*:
 $(v = v1 -- v2) \wedge (v2 = v3 -- v4) \wedge |lan\ v3| = x \longrightarrow |lan\ v| \geq x$
by (*metis le-add1 trans-le-add2 vertical-chop-add1*)

lemma *horizontal-chop-leq1*: $(v = u \| w) \longrightarrow u \leq v$
using *real-int.chop-leq1 hchop-def less-eq-view-ext-def order-refl* **by** *fastforce*

lemma *horizontal-chop-leq2*: $(v = u \| w) \longrightarrow w \leq v$
using *real-int.chop-leq2 hchop-def less-eq-view-ext-def order-refl* **by** *fastforce*

lemma *vertical-chop-leq1*: $(v = u -- w) \longrightarrow u \leq v$
using *nat-int.chop-subset1 vchop-def less-eq-view-ext-def order-refl* **by** *fastforce*

lemma *vertical-chop-leq2*: $(v = u -- w) \longrightarrow w \leq v$
using *nat-int.chop-subset2 vchop-def less-eq-view-ext-def order-refl* **by** *fastforce*

lemma *somewhere-leq*:
 $v \leq v' \iff (\exists\ v1\ v2\ v3\ vl\ vr\ vu\ vd.$
 $(v' = vl \| v1) \wedge (v1 = v2 \| vr) \wedge (v2 = vd -- v3) \wedge (v3 = v -- vu))$

proof
assume $v \leq v'$
hence *assm-exp*: $(ext\ v \leq ext\ v') \wedge (lan\ v \sqsubseteq lan\ v') \wedge (own\ v = own\ v')$
using *less-eq-view-ext-def* **by** *auto*
obtain *vl v1 v2 vr*
where
 $vl : vl = (\text{ext} = Abs-real-int(left(ext\ v'), left(ext\ v)), lan = lan\ v', own = own\ v')$
and
 $v1 : v1 = (\text{ext} = Abs-real-int(left(ext\ v), right(ext\ v')), lan = lan\ v', own = own\ v')$
and
 $v2 : v2 = (\text{ext} = Abs-real-int(left(ext\ v), right(ext\ v)), lan = lan\ v', own = own\ v')$

```

and
  vr:vr=(| ext=Abs-real-int(right(ext v),right(ext v')), lan=lan v', own=own v'|)
by blast
have vl-in-type:(left (ext v'), left (ext v)) ∈ {r::(real*real) . fst r ≤ snd r}
using less-eq-real-int-def assm-exp real-int.left-leq-right snd-conv
fst-conv mem-Collect-eq by simp
have v1-in-type:(left (ext v), right (ext v')) ∈ {r::(real*real) . fst r ≤ snd r}
using less-eq-real-int-def assm-exp real-int.left-leq-right snd-conv fst-conv
mem-Collect-eq order-trans by fastforce
have v2-in-type:(left (ext v), right (ext v)) ∈ {r::(real*real) . fst r ≤ snd r}
using less-eq-real-int-def assm-exp real-int.left-leq-right snd-conv fst-conv
mem-Collect-eq order-trans by fastforce
have vr-in-type:(right (ext v), right (ext v')) ∈ {r::(real*real) . fst r ≤ snd r}
using less-eq-real-int-def assm-exp real-int.left-leq-right snd-conv fst-conv
mem-Collect-eq order-trans by fastforce
then have hchops:(v'=vl||v1) ∧ (v1=v2||vr)
using vl v1 v2 vr less-eq-real-int-def hchop-def real-int.rchop-def
vl-in-type v1-in-type v2-in-type vr-in-type Abs-real-int-inverse by auto
have lanes-v2:lan v2 = lan v' using v2 by auto
have own-v2:own v2 = own v' using v2 by auto
have ext-v2:ext v2 = ext v
using v2 v2-in-type Abs-real-int-inverse by (simp add: Rep-real-int-inverse)
show
  ∃ v1 v2 v3 vl vr vu vd. (v'=vl||v1) ∧ (v1=v2||vr) ∧ (v2=vd--v3) ∧ (v3=v--vu)
proof (cases lan v' = lan v)
case True
obtain vd v3 vu
  where vd:vd = (| ext = ext v2, lan = ∅, own = own v'|)
  and v3:v3 = (| ext = ext v2, lan = lan v', own = own v'|)
  and vu:vu = (| ext = ext v2, lan = ∅, own = own v'|) by blast
hence (v2=vd--v3) ∧ (v3=v--vu)
using vd v3 vu True vchop-def nat-int.nchop-def nat-int.un-empty-absorb1
nat-int.un-empty-absorb2 nat-int.inter-empty1 nat-int.inter-empty2 lanes-v2
own-v2 ext-v2 assm-exp by auto
then show ?thesis using hchops by blast
next
case False
then have v'-neq-empty:lan v' ≠ ∅
  by (metis assm-exp nat-int.card-empty-zero nat-int.card-non-empty-geq-one
nat-int.card-subset-le le-0-eq)
show ?thesis
proof (cases lan v ≠ ∅)
case False
obtain vd v3 vu where vd:vd = (| ext = ext v2, lan = ∅, own = own v'|)
  and v3:v3 = (| ext = ext v2, lan = lan v', own = own v'|)
  and vu:vu = (| ext = ext v2, lan = lan v', own = own v'|) by blast
then have (v2=vd--v3) ∧ (v3=v--vu)
using vd v3 vu False vchop-def nat-int.nchop-def
nat-int.un-empty-absorb1 nat-int.un-empty-absorb2

```

```

    nat-int.inter-empty1 nat-int.inter-empty2 lanes-v2 own-v2 ext-v2 assm-exp
  by auto
then show ?thesis
  using hchops by blast
next
case True
show ?thesis
proof (cases (minimum (lan v)) = minimum (lan v'))
  assume min:minimum (lan v) = minimum (lan v')
  hence max:maximum (lan v) < maximum (lan v')
  by (metis Rep-nat-int-inverse assm-exp atLeastAtMost-subset-iff leI le-antisym
    nat-int.leq-max-sup nat-int.leq-min-inf nat-int.maximum-def nat-int.minimum-def
    nat-int.rep-non-empty-means-seq less-eq-nat-int.rep-eq False True
    v'-neq-empty)
  obtain vd v3 vu
    where vd:vd = (| ext = ext v2, lan =  $\emptyset$ , own = own v')
    and v3:v3 = (| ext = ext v2, lan = lan v', own = own v')
    and vu:vu = (| ext = ext v2, lan =
      Abs-nat-int({maximum(lan v)+1..maximum(lan v')}), own = own v')
    by blast
  have vu-in-type:
    {maximum(lan v)+1 ..maximum(lan v')}  $\in$  {S.( $\exists$  (m::nat) n.{m..n }=S)}
    using max by auto
  have consec:consec (lan v) (lan vu) using max
    by (simp add: Suc-leI nat-int.consec-def nat-int.leq-min-inf'
      nat-int.leq-nat-non-empty True vu)
  have disjoint:lan v  $\sqcap$  lan vu =  $\emptyset$ 
    by (simp add: consec nat-int.consec-inter-empty)
  have union:lan v' = lan v  $\sqcup$  lan vu
    by (metis Suc-eq-plus1 Suc-leI consec leq-max-sup' max min
      nat-int.consec-un-equality nat-int.consec-un-max nat-int.consec-un-min
      select-convs(2) v'-neq-empty vu)
  then have (v2=vd--v3)  $\wedge$  (v3=v--vu)
    using vd v3 vu vchop-def nat-int.nchop-def nat-int.un-empty-absorb1
      nat-int.un-empty-absorb2 nat-int.inter-empty1 nat-int.inter-empty2
      lanes-v2 own-v2 ext-v2 assm-exp vu-in-type Abs-nat-int-inverse
      consec union disjoint select-convs
    by force
  then show ?thesis using hchops by blast
next
assume (minimum (lan v))  $\neq$  minimum (lan v')
then have min:minimum (lan v) > minimum (lan v')
  by (metis Min-le True assm-exp finite-atLeastAtMost le-neq-implies-less
    less-eq-nat-int.rep-eq nat-int.el.rep-eq nat-int.minimum-def
    nat-int.minimum-in rep-non-empty-means-seq subsetCE v'-neq-empty)
show ?thesis
proof (cases (maximum (lan v)) = maximum (lan v'))
  assume max:maximum(lan v) = maximum (lan v')
  obtain vd v3 vu

```

```

where
  vd:vd =
    (| ext = ext v2,
      lan = Abs-nat-int ({minimum(lan v')..minimum(lan v)-1}),
      own = own v'|)
and
  v3:v3 = (| ext = ext v2, lan = lan v, own = own v' |)
and
  vu:vu = (| ext = ext v2, lan =  $\emptyset$ , own = own v' |) by blast
have consec:consec (lan vd) (lan v)
using True leq-max-sup' leq-nat-non-empty min
  nat-int.consec-def vd by auto
have maximum (lan vd  $\sqcup$  lan v) = maximum (lan v)
using consec consec-un-max by auto
then have max':maximum (lan vd  $\sqcup$  lan v) = maximum (lan v')
by (simp add: max)
have minimum (lan vd  $\sqcup$  lan v) = minimum (lan vd)
using consec consec-un-min by auto
then have min':minimum (lan vd  $\sqcup$  lan v) = minimum (lan v')
by (metis atLeastatMost-empty-iff vd bot-nat-int.abs-eq consec
  nat-int.consec-def nat-int.leq-min-inf' select-convs(2))
have union: lan v' = lan vd  $\sqcup$  lan v
using consec max' min' nat-int.consec-un-equality v'-neq-empty
by fastforce
then have (v2=vd--v3)  $\wedge$  (v3=v--vu)
using assm-exp consec ext-v2 lanes-v2 nat-int.nchop-def nat-int.un-empty-absorb1
  own-v2 v3 vd view.vchop-def vu by force
then show ?thesis
using hchops by blast
next
assume (maximum (lan v))  $\neq$  maximum (lan v')
then have max:maximum (lan v) < maximum (lan v')
by (metis assm-exp atLeastatMost-subset-iff nat-int.leq-max-sup
  nat-int.maximum-def nat-int.rep-non-empty-means-seq less-eq-nat-int.rep-eq
  True order.not-eq-order-implies-strict v'-neq-empty)
obtain vd v3 vu
where vd:
  vd = (| ext = ext v2,
        lan = Abs-nat-int ({minimum(lan v')..minimum(lan v)-1}),
        own = own v'|)
and v3:
  v3 = (| ext = ext v2, lan = lan v  $\sqcup$  lan vu, own = own v' |)
and vu:
  vu = (| ext = ext v2,
        lan = Abs-nat-int ({maximum(lan v)+1..maximum(lan v')})),
        own = own v' |) by blast
have consec:consec (lan v) (lan vu)
using True leq-nat-non-empty max nat-int.consec-def nat-int.leq-min-inf'
  vu

```



```

    by auto
  have union:lan v3 = lan v  $\sqcup$  lan vu
    by (simp add: v3 min max consec)
  then have chop1: (v3=v--vu)
    using assm-exp consec ext-v2 nat-int.nchop-def v3 view.vchop-def
      vu
    by auto
  have min-eq:minimum (lan v3) = minimum (lan v)
    using chop1 consec nat-int.chop-min vchop-def by blast
  have neq3:lan v3  $\neq$   $\emptyset$ 
  by (metis bot.extremum-uniqueI consec nat-int.consec-def nat-int.un-subset2
    union)
  have consec2:consec (lan vd) (lan v3)
  using min consec union min-eq Suc-leI nat-int.consec-def nat-int.leq-max-sup'
    nat-int.leq-min-inf' nat-int.leq-nat-non-empty neq3 v3 vd
    by (auto)
  have minimum (lan vd  $\sqcup$  lan v3) = minimum (lan vd)
    using consec2 consec-un-min by auto
  then have min':minimum (lan vd  $\sqcup$  lan v3) = minimum (lan v')
  by (metis vd atLeastatMost-empty-iff2 bot-nat-int.abs-eq consec2 leq-min-inf'
    nat-int.consec-def select-convs(2))
  have maximum (lan vd  $\sqcup$  lan v3) = maximum (lan v3)
    using consec2 consec-un-max by auto
  then have maximum (lan vd  $\sqcup$  lan v3) = maximum (lan vu)
    using consec consec-un-max union by auto
  then have max':maximum (lan vd  $\sqcup$  lan v3) = maximum (lan v')
    by (metis Suc-eq-plus1 Suc-leI max nat-int.leq-max-sup'
      select-convs(2) vu)
  have union2: lan v' = lan vd  $\sqcup$  lan v3
    using min max consec2 neq3 min' max' nat-int.consec-un-equality
v'-neq-empty
    by force
  have (v2=vd--v3)  $\wedge$  (v3=v--vu)
    using union2 chop1 consec2 nat-int.nchop-def v2 v3 vd
      view.vchop-def
    by fastforce
  then show ?thesis using hchops by blast
qed
qed
qed
qed
next
assume
 $\exists v1 v2 v3 vl vr vu vd. (v'=vl||v1) \wedge (v1=v2||vr) \wedge (v2=vd--v3) \wedge (v3=v--vu)$ 
then obtain v1 v2 v3 vl vr vu vd
  where (v'=vl||v1)  $\wedge$  (v1=v2||vr)  $\wedge$  (v2=vd--v3)  $\wedge$  (v3=v--vu) by blast
then show  $v \leq v'$ 
  by (meson horizontal-chop-leq1 horizontal-chop-leq2 order-trans vertical-chop-leq1
    vertical-chop-leq2)

```

qed

The switch relation is compatible with the chopping relations, in the following sense. If we can chop a view v into two subviews u and w , and we can reach v' via the switch relation, then there also exist two subviews u' , w' of v' , such that u' is reachable from u (and respectively for w' , w).

lemma *switch-unique*: $(v = c > u) \wedge (v = c > w) \longrightarrow u = w$
using *switch-def* **by** *auto*

lemma *switch-exists*: $\exists c u. (v = c > u)$
using *switch-def* **by** *auto*

lemma *switch-always-exists*: $\forall c. \exists u. (v = c > u)$
by (*metis select-convs switch-def*)

lemma *switch-origin*: $\exists u. (u = (\text{own } v) > v)$
using *switch-def* **by** *auto*

lemma *switch-refl*: $(v = (\text{own } v) > v)$
by (*simp add: switch-def*)

lemma *switch-symm*: $(v = c > u) \longrightarrow (u = (\text{own } v) > v)$
by (*simp add: switch-def*)

lemma *switch-trans*: $(v = c > u) \wedge (u = d > w) \longrightarrow (v = d > w)$
by (*simp add: switch-def*)

lemma *switch-triangle*: $(v = c > u) \wedge (v = d > w) \longrightarrow (u = d > w)$
using *switch-def* **by** *auto*

lemma *switch-hchop1*:
 $(v = v1 \parallel v2) \wedge (v = c > v')$ \longrightarrow
 $(\exists v1' v2'. (v1 = c > v1') \wedge (v2 = c > v2') \wedge (v' = v1' \parallel v2'))$
by (*metis (no-types, opaque-lifting) select-convs view.hchop-def view.switch-def*)

lemma *switch-hchop2*:
 $(v' = v1' \parallel v2') \wedge (v = c > v')$ \longrightarrow
 $(\exists v1 v2. (v1 = c > v1') \wedge (v2 = c > v2') \wedge (v = v1 \parallel v2))$
by (*metis (no-types, opaque-lifting) select-convs view.hchop-def view.switch-def*)

lemma *switch-vc Chop1*:
 $(v = v1 \dashv\vdash v2) \wedge (v = c > v')$ \longrightarrow
 $(\exists v1' v2'. (v1 = c > v1') \wedge (v2 = c > v2') \wedge (v' = v1' \dashv\vdash v2'))$
by (*metis (no-types, opaque-lifting) select-convs view.vchop-def view.switch-def*)

lemma *switch-vc Chop2*:
 $(v' = v1' \dashv\vdash v2') \wedge (v = c > v')$ \longrightarrow
 $(\exists v1 v2. (v1 = c > v1') \wedge (v2 = c > v2') \wedge (v = v1 \dashv\vdash v2))$
by (*metis (no-types, opaque-lifting) select-convs view.vchop-def view.switch-def*)

```

lemma switch-leq:  $u' \leq u \wedge (v=c>u) \longrightarrow (\exists v'. (v'=c>u') \wedge v' \leq v)$ 
proof
  assume assm:  $u' \leq u \wedge (v=c>u)$ 
  then have more-eq:  $\text{more } v = \text{more } u$ 
    using view.switch-def by blast
  then obtain v' where v'-def:  $v' = (\text{ext} = \text{ext } u', \text{lan} = \text{lan } u', \text{own} = \text{own } v)$ 
    by blast
  have ext:  $\text{ext } v' \leq \text{ext } v$  using assm switch-def
    by (simp add: less-eq-view-ext-def v'-def)
  have lan:  $\text{lan } v' \leq \text{lan } v$  using assm switch-def
    by (simp add: less-eq-view-ext-def v'-def)
  have more:  $\text{more } v' \leq \text{more } v$  using more-eq assm by simp
  have less:  $v' \leq v$  using less-eq-view-ext-def ext lan more v'-def
    by (simp add: less-eq-view-ext-def)
  have switch:  $v' = c > u'$  using v'-def switch-def assm
    by (simp add: less-eq-view-ext-def)
  show  $(\exists v'. (v' = c > u') \wedge v' \leq v)$  using switch less by blast
qed
end
end

```

6 Restrict Claims and Reservations to a View

To model that a view restricts the number of lanes a car may perceive, we define a function *restrict* taking a view v , a function f from cars to lanes and a car c , and returning the intersection between $f(c)$ and the set of lanes of v . This function will in the following only be applied to the functions yielding reservations and claims from traffic snapshots.

The lemmas of this section describe the connection between *restrict* and the different operations on traffic snapshots and views (e.g., the transition relations or the fact that reservations and claims are consecutive).

theory *Restriction*

imports *Traffic Views*

begin

locale *restriction* = *view+traffic*

begin

definition *restrict* :: $\text{view} \Rightarrow (\text{cars} \Rightarrow \text{lanes}) \Rightarrow \text{cars} \Rightarrow \text{lanes}$

where $\text{restrict } v f c == (f c) \sqcap \text{lan } v$

lemma *restrict-def'*: $\text{restrict } v f c = \text{lan } v \sqcap f c$

using *inf-commute* *restriction.restrict-def* **by** *auto*

lemma *restrict-subseteq*: $\text{restrict } v f c \sqsubseteq f c$

using *inf-le1 restrict-def* **by** *auto*

lemma *restrict-clm* : $\text{restrict } v \text{ (clm } ts) \ c \sqsubseteq \text{clm } ts \ c$
using *inf-le1 restrict-def* **by** *auto*

lemma *restrict-res*: $\text{restrict } v \text{ (res } ts) \ c \sqsubseteq \text{res } ts \ c$
using *inf-le1 restrict-def* **by** *auto*

lemma *restrict-view*: $\text{restrict } v \ f \ c \sqsubseteq \text{lan } v$
using *inf-le1 restrict-def* **by** *auto*

lemma *restriction-stable*: $(v=u||w) \longrightarrow \text{restrict } u \ f \ c = \text{restrict } w \ f \ c$
using *hchop-def restrict-def* **by** *auto*

lemma *restriction-stable1*: $(v=u||w) \longrightarrow \text{restrict } v \ f \ c = \text{restrict } u \ f \ c$
by (*simp add: hchop-def restrict-def*)

lemma *restriction-stable2*: $(v=u||w) \longrightarrow \text{restrict } v \ f \ c = \text{restrict } w \ f \ c$
by (*simp add: restriction-stable restriction-stable1*)

lemma *restriction-un*:
 $(v=u--w) \longrightarrow \text{restrict } v \ f \ c = (\text{restrict } u \ f \ c \sqcup \text{restrict } w \ f \ c)$
using *nat-int.inter-distr1 nat-int.inter-empty1 nat-int.un-empty-absorb1*
nat-int.un-empty-absorb2 nat-int.nchop-def restrict-def vchop-def
by *auto*

lemma *restriction-mon1*: $(v=u--w) \longrightarrow \text{restrict } u \ f \ c \sqsubseteq \text{restrict } v \ f \ c$
using *inf-mono nat-int.chop-subset1 restrict-def vchop-def*
by (*metis (no-types, opaque-lifting) order-refl*)

lemma *restriction-mon2*: $(v=u--w) \longrightarrow \text{restrict } w \ f \ c \sqsubseteq \text{restrict } v \ f \ c$
using *inf-mono nat-int.chop-subset2 restrict-def vchop-def*
by (*metis (no-types, opaque-lifting) order-refl*)

lemma *restriction-disj*: $(v=u--w) \longrightarrow (\text{restrict } u \ f \ c) \sqcap (\text{restrict } w \ f \ c) = \emptyset$
proof
assume *assm*: $v=u--w$
then have $1:\text{lan } u \sqcap \text{lan } w = \emptyset$ **using** *vchop-def*
by (*metis inf-commute inter-empty1 nat-int.consec-inter-empty nat-int.nchop-def*)
from *assm* **have** $(\text{restrict } u \ f \ c) \sqcap (\text{restrict } w \ f \ c) \sqsubseteq \text{lan } u \sqcap \text{lan } w$
by (*meson inf-mono restriction.restrict-view*)
with 1 **show** $(\text{restrict } u \ f \ c) \sqcap (\text{restrict } w \ f \ c) = \emptyset$
by (*simp add: bot.extremum-uniqueI*)

qed

lemma *vertical-chop-restriction-res-consec-or-empty*:
 $(v=v1--v2) \wedge \text{restrict } v1 \text{ (res } ts) \ c \neq \emptyset \wedge \text{consec } ((\text{lan } v1)) ((\text{lan } v2)) \wedge$
 $\neg \text{consec } (\text{restrict } v1 \text{ (res } ts) \ c) (\text{restrict } v2 \text{ (res } ts) \ c)$
 $\longrightarrow \text{restrict } v2 \text{ (res } ts) \ c = \emptyset$

proof

assume *assm*:

$(v=v1--v2) \wedge \text{restrict } v1 \text{ (res ts) } c \neq \emptyset \wedge$
 $\text{consec } ((\text{lan } v1)) ((\text{lan } v2)) \wedge$

$\neg \text{consec } (\text{restrict } v1 \text{ (res ts) } c) (\text{restrict } v2 \text{ (res ts) } c)$

hence $\text{restrict } v1 \text{ (res ts) } c = \emptyset \vee \text{restrict } v2 \text{ (res ts) } c = \emptyset \vee$

$(\text{maximum } (\text{restrict } v1 \text{ (res ts) } c)+1 \neq \text{minimum } (\text{restrict } v2 \text{ (res ts) } c))$

using *nat-int.consec-def* **by** *blast*

hence *empty-or-non-consec:restrict v2 (res ts) c = $\emptyset \vee$*

(maximum (restrict v1 (res ts) c)+1 \neq minimum (restrict v2 (res ts) c))

using *assm* **by** *blast*

have *consec-lanes:consec ((lan v1)) ((lan v2))* **by** (*simp add: assm*)

have *subs:restrict v2 (res ts) c \sqsubseteq ((lan v2))* **using** *restrict-view*

by *simp*

show *restrict v2 (res ts) c = \emptyset*

proof (*rule ccontr*)

assume *non-empty:restrict v2 (res ts) c $\neq \emptyset$*

hence *max*:

$(\text{maximum } (\text{restrict } v1 \text{ (res ts) } c)+1 \neq \text{minimum } (\text{restrict } v2 \text{ (res ts) } c))$

using *empty-or-non-consec* **by** *blast*

have *ex-n: $\exists n. n \in \text{Rep-nat-int } (\text{restrict } v2 \text{ (res ts) } c)$*

using *nat-int.el.rep-eq non-empty nat-int.non-empty-elem-in* **by** *auto*

have *res1-or2:|(res ts) c| = 1 \vee |(res ts) c| = 2*

by (*metis Suc-1 atLeastOneRes atMostTwoRes dual-order.antisym le-SucE*)

then show *False*

proof

assume *res-one:|(res ts) c|=1*

then obtain *n* **where** *one-lane:Rep-nat-int ((res ts) c) = {n}*

using *singleton* **by** *blast*

then have *n \in Rep-nat-int (restrict v1 (res ts) c)*

by (*metis assm equals0D nat-int.el.rep-eq less-eq-nat-int.rep-eq*
nat-int.non-empty-elem-in restrict-res singletonI subset-singletonD)

then have *Rep-nat-int (restrict v2 (res ts) c) = {}*

by (*metis one-lane assm inf.absorb1 less-eq-nat-int.rep-eq restriction.restrict-res*
restriction-disj subset-singleton-iff)

thus *False*

using *ex-n* **by** *blast*

next

assume *res-two:|(res ts) c| = 2*

hence *ex-two-ln: ($\exists n. \text{Rep-nat-int } ((\text{res ts) } c) = \{n, n+1\}$)*

using *consecutiveRes* **by** *blast*

then obtain *n* **where** *n-def:Rep-nat-int ((res ts) c) = {n, n+1}* **by** *blast*

hence *rep-restrict-v1:Rep-nat-int (restrict v1 (res ts) c) \subseteq {n, n+1}*

using *less-eq-nat-int.rep-eq restrict-res* **by** *blast*

hence

n \in Rep-nat-int (restrict v1 (res ts) c) \vee

n+1 \in Rep-nat-int (restrict v1 (res ts) c)

using *assm bot.extremum-unique less-eq-nat-int.rep-eq* **by** *fastforce*

thus *False*

proof
assume $\text{suc-n-in-res-v1:n+1} \in \text{Rep-nat-int} (\text{restrict } v1 \text{ (res ts) } c)$
hence $\text{suc-n-in-v1:n+1} \in \text{Rep-nat-int} ((\text{lan } v1))$
using $\text{less-eq-nat-int.rep-eq restrict-view}$ **by** blast
hence $n+1 \notin \text{Rep-nat-int} (\text{lan } v2)$
using $\text{assm vchop-def nat-int.nchop-def nat-int.consec-in-exclusive1}$
 $\text{nat-int.el.rep-eq nat-int.not-in.rep-eq}$ **by** blast
hence $\text{suc-n-not-in-res-v2:n+1} \notin \text{Rep-nat-int} (\text{restrict } v2 \text{ (res ts) } c)$
using $\text{less-eq-nat-int.rep-eq subs}$ **by** blast
have $\forall m . m \in \text{lan } v2 \longrightarrow m \geq \text{minimum} (\text{lan } v2)$
by $(\text{metis consec-lanes nat-int.minimum-def nat-int.consec-def}$
 $\text{nat-int.el.rep-eq atLeastAtMost-iff nat-int.leq-min-inf}$
 $\text{nat-int.rep-non-empty-means-seq})$
then have $\forall m . m \in \text{lan } v2 \longrightarrow m > \text{maximum} (\text{lan } v1)$
using $\text{assm nat-int.consec-def}$ **by** fastforce
then have $\forall m . m \in \text{lan } v2 \longrightarrow m > n+1$
using $\text{consec-lanes nat-int.maximum-def nat-int.card-seq}$
 $\text{nat-int.consec-def suc-n-in-v1}$
by $(\text{simp add: nat-int.consec-lesser})$
then have $n \notin \text{Rep-nat-int} ((\text{lan } v2))$
using $\text{suc-n-in-v1 assm nat-int.consec-def nat-int.el.rep-eq}$
by auto
hence $n \notin \text{Rep-nat-int} (\text{restrict } v2 \text{ (res ts) } c)$
using $\text{less-eq-nat-int.rep-eq rev-subsetD subs}$ **by** blast
hence $\text{Rep-nat-int} (\text{restrict } v2 \text{ (res ts) } c) = \{\}$
using $\text{insert-absorb insert-ident insert-not-empty n-def less-eq-nat-int.rep-eq}$
 $\text{restrict-res singleton-insert-inj-eq subset-insert suc-n-not-in-res-v2}$
by fastforce
thus False **using** ex-n **by** blast
next
assume $n\text{-in-res-v1:n} \in \text{Rep-nat-int} (\text{restrict } v1 \text{ (res ts) } c)$
hence $n\text{-not-in-v2:n} \notin \text{Rep-nat-int} ((\text{lan } v2))$
using $\text{assm vchop-def nat-int.nchop-def nat-int.consec-in-exclusive1}$
 $\text{nat-int.consec-in-exclusive2 nat-int.el.rep-eq nat-int.not-in.rep-eq}$
by $(\text{meson less-eq-nat-int.rep-eq restrict-view subsetCE})$
hence $n\text{-not-in-res-v2:n} \notin \text{Rep-nat-int} (\text{restrict } v2 \text{ (res ts) } c)$
using $\text{less-eq-nat-int.rep-eq subs}$ **by** blast
show False
proof $(\text{cases } n+1 \in \text{Rep-nat-int}(\text{restrict } v2 \text{ (res ts) } c))$
case False
hence $\text{Rep-nat-int} (\text{restrict } v2 \text{ (res ts) } c) = \{\}$
using $\text{insert-absorb insert-ident insert-not-empty n-def n-not-in-res-v2}$
 $\text{less-eq-nat-int.rep-eq restrict-res singleton-insert-inj-eq' subset-insert}$
by fastforce
thus False **using** ex-n **by** blast
next
case True
obtain $\text{NN} :: \text{nat set} \Rightarrow \text{nat} \Rightarrow \text{nat set}$ **where**
 $\forall x0 \ x1 . (\exists v2 . x0 = \text{insert } x1 \ v2 \wedge x1 \notin v2)$

$$= (x0 = \text{insert } x1 \text{ (NN } x0 \text{ } x1) \wedge x1 \notin \text{NN } x0 \text{ } x1)$$

by *moura*
then have *f1*:

$$\text{Rep-nat-int (restrict } v2 \text{ (res ts) } c) =$$

$$\text{insert (n + 1) (NN (Rep-nat-int (restrict } v2 \text{ (res ts) } c)) (n + 1))}$$

$$\wedge n + 1 \notin \text{NN (Rep-nat-int (restrict } v2 \text{ (res ts) } c)) (n + 1)}$$
by (*meson mk-disjoint-insert True*)
then have

$$\text{insert (n + 1) (NN (Rep-nat-int (restrict } v2 \text{ (res ts) } c)) (n+1))}$$

$$\subseteq \{n+1\} \cup \{\}$$
by (*metis (no-types) insert-is-Un n-def n-not-in-res-v2 less-eq-nat-int.rep-eq restrict-res subset-insert*)
then have $\{n + 1\} = \text{Rep-nat-int (restrict } v2 \text{ (res ts) } c)$
using *f1* **by** *blast*
then have *min:nat-int.minimum (restrict } v2 \text{ (res ts) } c) = n+1*
by (*metis (no-types) Min-singleton nat-int.minimum-def non-empty*)

then have *suc-n-in-v2:n+1* $\in (\text{lan } v2)$
using *nat-int.el.rep-eq less-eq-nat-int.rep-eq subs True*
by *auto*
have $\forall m . m \in \text{lan } v1 \longrightarrow m \leq \text{maximum (lan } v1)$
by (*metis atLeastAtMost-iff consec-lanes nat-int.maximum-def nat-int.consec-def nat-int.el.rep-eq nat-int.leq-max-sup nat-int.rep-non-empty-means-seq*)
then have $\forall m . m \in \text{lan } v1 \longrightarrow m < \text{minimum (lan } v2)$ **using** *assm*
using *nat-int.consec-lesser nat-int.minimum-in nat-int.consec-def*
by *auto*
then have $\forall m . m \in \text{lan } v1 \longrightarrow m < n+1$
using *consec-lanes nat-int.card-seq nat-int.consec-def suc-n-in-v2 nat-int.consec-lesser* **by** *auto*
then have *suc-n-not-in-v1:n+1* $\notin \text{Rep-nat-int ((lan } v1))$
using *nat-int.el.rep-eq* **by** *auto*
have *suc-n-not-in-res-v1:n+1* $\notin \text{Rep-nat-int (restrict } v1 \text{ (res ts) } c)$
using *less-eq-nat-int.rep-eq restrict-view suc-n-not-in-v1* **by** *blast*
from *rep-restrict-v1* **and** *n-in-res-v1* **have** *res-v1-singleton*:

$$\text{Rep-nat-int (restrict } v1 \text{ (res ts) } c) = \{n\}$$
using *Set.set-insert insert-absorb2 insert-commute singleton-insert-inj-eq' subset-insert suc-n-not-in-res-v1* **by** *blast*
have *max: nat-int.maximum (restrict } v1 \text{ (res ts) } c) = n*
by (*metis Rep-nat-int-inverse nat-int.leq-max-sup' order-refl res-v1-singleton nat-int.rep-single*)
from *min* **and** *max* **have**

$$\text{maximum (restrict } v1 \text{ (res ts) } c)+1 = \text{minimum (restrict } v2 \text{ (res ts) } c)$$
by *auto*
thus *False*
using *empty-or-non-consec non-empty* **by** *blast*
qed
qed
qed
qed

qed

lemma *restriction-consec-res:(v=u--w)*
→ *restrict u (res ts) c = ∅ ∨ restrict w (res ts) c = ∅*
∨ *consec (restrict u (res ts) c) (restrict w (res ts) c)*
proof
assume *assm:v=u--w*
then show *restrict u (res ts) c = ∅ ∨ restrict w (res ts) c = ∅*
∨ *consec (restrict u (res ts) c) (restrict w (res ts) c)*
proof (*cases res ts c = ∅*)
case *True*
show *?thesis*
by (*simp add: True inter-empty1 restriction.restrict-def'*)
next
case *False*
then have *res ts c ≠ ∅* by *best*
show *?thesis*
by (*metis (no-types, lifting) assm inter-empty1 nat-int.nchop-def*
restriction.restrict-def restriction.vertical-chop-restriction-res-consec-or-empty
view.vchop-def)

qed

qed

lemma *restriction-clm-res-disjoint:*
(restrict v (res ts) c) ∩ (restrict v (clm ts) c) = ∅
by (*metis (no-types) inf-assoc nat-int.inter-empty2 restriction.restrict-def*
restrict-def' traffic.disjoint)

lemma *el-in-restriction-clm-singleton:*
n ∈ restrict v (clm ts) c → (clm ts) c = Abs-nat-int({n})
proof
assume *n-in-restr:n ∈ restrict v (clm ts) c*
hence *n ∈ ((clm ts) c ∩ (lan v))* by (*simp add: restrict-def*)
hence *n ∈ (Rep-nat-int (clm ts c) ∩ Rep-nat-int (lan v))*
by (*simp add: inf-nat-int.rep-eq nat-int.el-def*)
hence *n-in-rep-clm:n ∈ (Rep-nat-int ((clm ts) c))* by *simp*
then have *(clm ts) c ≠ ∅* using *nat-int.el.rep-eq nat-int.non-empty-elem-in*
by *auto*
then have $|(clm ts) c| ≥ 1$
by (*simp add: nat-int.card-non-empty-geq-one*)
then have $|(clm ts) c| = 1$ using *atMostOneCln le-antisym*
by *blast*
with *n-in-rep-clm* show *(clm ts) c = Abs-nat-int({n})*
by (*metis Rep-nat-int-inverse nat-int.singleton singletonD*)

qed

lemma *restriction-clm-v2-non-empty-v1-empty:*
(v=u--w) ∧ restrict w (clm ts) c ≠ ∅ ∧
consec ((lan u)) ((lan w)) → restrict u (clm ts) c = ∅

proof

assume $assm:(v=u--w) \wedge restrict\ w\ (clm\ ts)\ c \neq \emptyset \wedge consec\ (lan\ u)\ (lan\ w)$
obtain n **where** $n\text{-in-res-v2}:(n \in restrict\ w\ (clm\ ts)\ c)$
using $assm\ maximum\text{-in}$ **by** $blast$
have $(clm\ ts)\ c = Abs\text{-nat-int}(\{n\})$
using $n\text{-in-res-v2}$ **by** $(simp\ add:\ el\text{-in-restriction-clm-singleton})$
then show $restrict\ u\ (clm\ ts)\ c = \emptyset$
by $(metis\ assm\ inf.\text{absorb-iff2}\ inf\text{-commute}\ less\text{-eq-nat-int.rep-eq}\ n\text{-in-res-v2}\ nat\text{-int.el.rep-eq}\ nat\text{-int.in-singleton}\ restriction.\text{restrict-subseteq}\ restriction.\text{restriction-disj}\ subsetI)$

qed

lemma $restriction\text{-consec-clm}$:

$(v=u--w) \wedge consec\ (lan\ u)\ (lan\ w)$
 $\longrightarrow restrict\ u\ (clm\ ts)\ c = \emptyset \vee restrict\ w\ (clm\ ts)\ c = \emptyset$
using $nat\text{-int.card-un-add}\ nat\text{-int.card-empty-zero}\ restriction\text{-un}\ atMostOneClm$
 $nat\text{-int.chop-add1}\ nat\text{-int.inter-distr1}\ nat\text{-int.inter-empty1}\ nat\text{-int.un-empty-absorb1}$
 $nat\text{-int.un-empty-absorb2}\ nat\text{-int.nchop-def}\ restrict\text{-def}\ vchop\text{-def}$
 $restriction\text{-clm-v2-non-empty-v1-empty}$
by $meson$

lemma $restriction\text{-add-res}$:

$(v=u--w)$
 $\longrightarrow |restrict\ v\ (res\ ts)\ c| = |restrict\ u\ (res\ ts)\ c| + |restrict\ w\ (res\ ts)\ c|$

proof

assume $assm: v=u--w$
then have $1: restrict\ u\ (res\ ts)\ c \sqcap restrict\ w\ (res\ ts)\ c = \emptyset$
using $restriction.\text{restriction-disj}$ **by** $auto$
from $assm$ **have** $restrict\ v\ (res\ ts)\ c = restrict\ u\ (res\ ts)\ c \sqcup restrict\ w\ (res\ ts)\ c$
using $restriction.\text{restriction-un}$ **by** $blast$
with 1 **show** $|restrict\ v\ (res\ ts)\ c| = |restrict\ u\ (res\ ts)\ c| + |restrict\ w\ (res\ ts)\ c|$
by $(metis\ add.\text{right-neutral}\ add\text{-cancel-left-left}\ assm\ nat\text{-int.card-empty-zero}\ nat\text{-int.card-un-add}\ nat\text{-int.un-empty-absorb1}\ nat\text{-int.un-empty-absorb2}\ restriction.\text{restriction-consec-res})$

qed

lemma $restriction\text{-eq-view-card}$: $restrict\ v\ f\ c = lan\ v \longrightarrow |restrict\ v\ f\ c| = |lan\ v|$
by $simp$

lemma $restriction\text{-card-mon1}$: $(v=u--w) \longrightarrow |restrict\ u\ f\ c| \leq |restrict\ v\ f\ c|$
using $nat\text{-int.card-subset-le}\ restriction\text{-mon1}$ **by** $(simp)$

lemma $restriction\text{-card-mon2}$: $(v=u--w) \longrightarrow |restrict\ w\ f\ c| \leq |restrict\ v\ f\ c|$
using $nat\text{-int.card-subset-le}\ restriction\text{-mon2}$ **by** $(simp)$

lemma $restriction\text{-res-leq-two}$: $|restrict\ v\ (res\ ts)\ c| \leq 2$
using $atMostTwoRes\ nat\text{-int.card-subset-le}\ le\text{-trans}\ restrict\text{-res}$

by *metis*

lemma *restriction-clm-leq-one*: $|restrict\ v\ (clm\ ts)\ c| \leq 1$
using *atMostOneClm nat-int.card-subset-le le-trans restrict-clm*
by *metis*

lemma *restriction-add-clm*:

($v=u--w$)

$\longrightarrow |restrict\ v\ (clm\ ts)\ c| = |restrict\ u\ (clm\ ts)\ c| + |restrict\ w\ (clm\ ts)\ c|$

proof

assume *assm*: $v=u--w$

have $|restrict\ u\ (clm\ ts)\ c| = 1 \vee |restrict\ u\ (clm\ ts)\ c| = 0$

by (*metis One-nat-def le-0-eq le-SucE restriction.restriction-clm-leq-one*)

then show $|restrict\ v\ (clm\ ts)\ c| = |restrict\ u\ (clm\ ts)\ c| + |restrict\ w\ (clm\ ts)\ c|$

proof

assume *clm-u*: $|restrict\ u\ (clm\ ts)\ c| = 1$

have $restrict\ u\ (clm\ ts)\ c \neq \emptyset$

using *clm-u card-non-empty-geq-one* by *auto*

then have $|restrict\ w\ (clm\ ts)\ c| = 0$

by (*metis (no-types) assm card-empty-zero inf-commute inter-empty2*
nat-int.nchop-def restriction.restrict-def
restriction.restriction-clm-v2-non-empty-v1-empty view.vchop-def)

then show *?thesis*

by (*metis Nat.add-0-right assm clm-u inf.absorb1 inf-commute*
restriction.restriction-card-mon1 restriction.restriction-clm-leq-one)

next

assume *no-clm-u*: $|restrict\ u\ (clm\ ts)\ c| = 0$

then have $|restrict\ w\ (clm\ ts)\ c| = 1 \vee |restrict\ w\ (clm\ ts)\ c| = 0$

by (*metis One-nat-def le-0-eq le-SucE restriction.restriction-clm-leq-one*)

then show *?thesis*

proof

assume $|restrict\ w\ (clm\ ts)\ c| = 1$

then show *?thesis*

by (*metis no-clm-u add-cancel-left-left antisym-conv3 assm leD*
restriction.restriction-card-mon2 restriction.restriction-clm-leq-one)

next

assume *no-clm-w*: $|restrict\ w\ (clm\ ts)\ c| = 0$

then have $|restrict\ v\ (clm\ ts)\ c| = 0$

by (*metis assm card-empty-zero chop-empty*
nat-int.card-non-empty-geq-one nat-int.nchop-def no-clm-u
restriction.restriction-un)

then show *?thesis*

using *no-clm-u no-clm-w* by *linarith*

qed

qed

qed

lemma *restriction-card-mon-trans*:

$(v=v1--v2) \wedge (v2=v3--v4) \wedge |restrict\ v3\ f\ c| = 1 \longrightarrow |restrict\ v\ f\ c| \geq 1$
by (*metis One-nat-def Suc-leI neq0-conv not-less-eq-eq*
restriction-card-mon1 restriction-card-mon2)

lemma *restriction-card-somewhere-mon*:

$(v=vl||v1) \wedge (v1=v2||vr) \wedge (v2=vu--v3) \wedge (v3=v'--vd) \wedge |restrict\ v'\ f\ c|=1$
 $\longrightarrow |restrict\ v\ f\ c| \geq 1$

proof

assume *assm*:

$(v=vl||v1) \wedge (v1=v2||vr) \wedge (v2=vu--v3) \wedge (v3=v'--vd) \wedge |restrict\ v'\ f\ c|=1$

then have $|restrict\ v2\ f\ c| \geq 1$ **using** *restriction-card-mon-trans* **by** *blast*

then show $|restrict\ v\ f\ c| \geq 1$ **using** *restriction-stable1 restriction-stable2 assm*

by *metis*

qed

lemma *restrict-eq-lan-subst*:

$|restrict\ v\ f\ c| = |lan\ v| \wedge (restrict\ v\ f\ c \sqsubseteq lan\ v) \longrightarrow restrict\ v\ f\ c = lan\ v$

proof

assume *assm*: $|restrict\ v\ f\ c| = |lan\ v| \wedge (restrict\ v\ f\ c \sqsubseteq lan\ v)$

have $|restrict\ v\ f\ c| = 0 \vee |restrict\ v\ f\ c| \neq 0$ **by** *auto*

thus $restrict\ v\ f\ c = lan\ v$

proof (*cases* $|restrict\ v\ f\ c| = 0$)

case *True*

then have *res-empty*: $restrict\ v\ f\ c = \emptyset$

by (*metis nat-int.card-non-empty-geq-one nat-int.card-empty-zero*)

then have $lan\ v = \emptyset$

by (*metis assm nat-int.card-empty-zero nat-int.card-non-empty-geq-one*)

then show $restrict\ v\ f\ c = lan\ v$

using *res-empty* **by** *auto*

next

case *False*

show $restrict\ v\ f\ c = lan\ v$

proof (*rule ccontr*)

assume *non-eq*: $restrict\ v\ f\ c \neq lan\ v$

then have $restrict\ v\ f\ c < lan\ v$

by (*simp add: assm less-le*)

then have $|restrict\ v\ f\ c| < |lan\ v|$

using *card-subset-less* **by** *blast*

then show *False* **using** *assm* **by** *simp*

qed

qed

qed

lemma *create-reservation-restrict-union*:

$(ts-r(c) \rightarrow ts')$

$\longrightarrow restrict\ v\ (res\ ts')\ c = restrict\ v\ (res\ ts)\ c \sqcup restrict\ v\ (clm\ ts)\ c$

proof

assume *assm*: $(ts-r(c) \rightarrow ts')$

```

hence  $res\text{-}ts':res\ ts' c = res\ ts\ c \sqcup clm\ ts\ c$ 
  by (simp add: create-reservation-def)
show  $restrict\ v\ (res\ ts')\ c = restrict\ v\ (res\ ts)\ c \sqcup restrict\ v\ (clm\ ts)\ c$ 
proof (cases clm ts c =  $\emptyset$ )
  case True
    hence  $res\text{-}ts'eq\text{-}ts:res\ ts' c = res\ ts\ c$ 
      using res-ts' nat-int.un-empty-absorb1 by simp
    from True have  $restrict\text{-}clm:restrict\ v\ (clm\ ts)\ c = \emptyset$ 
      using nat-int.inter-empty2 restrict-def by simp
    from res-ts'eq-ts have  $restrict\ v\ (res\ ts')\ c = restrict\ v\ (res\ ts)\ c$ 
      by (simp add: restrict-def)
    thus ?thesis using restrict-clm
      by (simp add: nat-int.un-empty-absorb1)
  next
    case False
      hence  $consec\ (clm\ ts\ c)\ (res\ ts\ c) \vee consec\ (res\ ts\ c)\ (clm\ ts\ c)$ 
        by (simp add: clm-consec-res)
      thus ?thesis
    proof
      assume  $consec:consec\ (clm\ ts\ c)\ (res\ ts\ c)$ 
      then show ?thesis
        using inter-distr1 res-ts' restriction.restrict-def
        by (simp add: Un-ac( $\exists$ ) inf-commute nat-int.union-def)
      next
        assume  $consec:consec\ (res\ ts\ c)\ (clm\ ts\ c)$ 
        then show ?thesis
          by (simp add: inter-distr2 res-ts' restriction.restrict-def)
    qed
  qed
qed

```

```

lemma switch-restrict-stable:( $v=c>u$ )  $\longrightarrow$  restrict\ v\ f\ d = restrict\ u\ f\ d
  using switch-def by (simp add: restrict-def)
end
end

```

7 Move a View according to Difference between Traffic Snapshots

In this section, we define a function to move a view according to the changes between two traffic snapshots. The intuition is that the view moves with the same speed as its owner. That is, if we move a view v from ts to ts' , we shift the extension of the view by the difference in the position of the owner of v .

```

theory Move
  imports Traffic Views

```

begin

context *traffic*
begin

definition *move::traffic* \Rightarrow *traffic* \Rightarrow *view* \Rightarrow *view*

where

$$\begin{aligned} \text{move } ts \ ts' \ v = & \langle \text{ext} = \text{shift } (\text{ext } v) ((\text{pos } ts' \ (\text{own } v)) - \text{pos } ts \ (\text{own } v)), \\ & \text{lan} = \text{lan } v, \\ & \text{own} = \text{own } v \rangle \end{aligned}$$

lemma *move-keeps-length*: $\|\text{ext } v\| = \|\text{ext } (\text{move } ts \ ts' \ v)\|$
using *real-int.shift-keeps-length* **by** (*simp add: move-def*)

lemma *move-keeps-lanes*: $\text{lan } v = \text{lan } (\text{move } ts \ ts' \ v)$ **using** *move-def* **by** *simp*

lemma *move-keeps-owner*: $\text{own } v = \text{own } (\text{move } ts \ ts' \ v)$ **using** *move-def* **by** *simp*

lemma *move-nothing* : $\text{move } ts \ ts \ v = v$ **using** *real-int.shift-zero* *move-def* **by** *simp*

lemma *move-trans*:

$(ts \Rightarrow ts') \wedge (ts' \Rightarrow ts'') \longrightarrow \text{move } ts' \ ts'' \ (\text{move } ts \ ts' \ v) = \text{move } ts \ ts'' \ v$

proof

assume *assm*: $(ts \Rightarrow ts') \wedge (ts' \Rightarrow ts'')$

have

$$\begin{aligned} & (\text{pos } ts'' \ (\text{own } v)) - \text{pos } ts \ (\text{own } v) \\ &= (\text{pos } ts'' \ (\text{own } v) + \text{pos } ts' \ (\text{own } v)) - (\text{pos } ts \ (\text{own } v) + \text{pos } ts' \ (\text{own } v)) \\ & \text{by } \textit{simp} \end{aligned}$$

have

$$\begin{aligned} & \text{move } ts' \ ts'' \ (\text{move } ts \ ts' \ v) = \\ & \langle \text{ext} = \\ & \text{shift } (\text{ext } (\text{move } ts \ ts' \ v)) \\ & (\text{pos } ts'' \ (\text{own } (\text{move } ts \ ts' \ v)) - \text{pos } ts' \ (\text{own } (\text{move } ts \ ts' \ v))), \\ & \text{lan} = \text{lan } (\text{move } ts \ ts' \ v), \\ & \text{own} = \text{own } (\text{move } ts \ ts' \ v) \rangle \end{aligned}$$

using *move-def* **by** *blast*

hence $\text{move } ts' \ ts'' \ (\text{move } ts \ ts' \ v) =$

$\langle \text{ext} = \text{shift } (\text{ext } (\text{move } ts \ ts' \ v)) (\text{pos } ts'' \ (\text{own } v) - \text{pos } ts' \ (\text{own } v)),$
 $\text{lan} = \text{lan } v, \text{own} = \text{own } v \rangle$

using *move-def* **by** *simp*

then show $\text{move } ts' \ ts'' \ (\text{move } ts \ ts' \ v) = \text{move } ts \ ts'' \ v$

proof –

have *f2*: $\forall x0 \ x1. (x1::\text{real}) + x0 = x0 + x1$

by *auto*

have

$$\begin{aligned} & \text{pos } ts'' \ (\text{own } v) \\ & + -1 * \text{pos } ts' \ (\text{own } v) + (\text{pos } ts' \ (\text{own } v) + -1 * \text{pos } ts \ (\text{own } v)) \\ &= \text{pos } ts'' \ (\text{own } v) + -1 * \text{pos } ts \ (\text{own } v) \end{aligned}$$

by *auto*

```

then have
  (shift (ext v) ((pos ts'' (own v)) + (-1 * pos ts (own v)))) =
  shift (shift (ext v) (pos ts' (own v) + -1 * pos ts (own v)))
  (pos ts'' (own v) + -1 * pos ts' (own v))
by (metis f2 real-int.shift-additivity)
then show ?thesis
using move-def f2 by simp
qed
qed

lemma move-stability-res:(ts-r(c)→ts') → move ts ts' v = v
and move-stability-clm:(ts-c(c,n)→ts') → move ts ts' v = v
and move-stability-wdr:(ts-wdr(c,n)→ts') → move ts ts' v = v
and move-stability-wdc:(ts-wdc(c)→ts') → move ts ts' v = v
using create-reservation-def create-claim-def withdraw-reservation-def
  withdraw-claim-def move-def move-nothing
by (auto)+

end
end

```

8 Sensors for Cars

This section presents the abstract definition of a function determining the sensor capabilities of cars. Such a function takes a car e , a traffic snapshot ts and another car c , and returns the length of c as perceived by e at the situation determined by ts . The only restriction we impose is that this length is always greater than zero.

With such a function, we define a derived notion of the *space* the car c occupies as perceived by e . However, this does not define the lanes c occupies, but only a continuous interval. The lanes occupied by c are given by the reservation and claim functions of the traffic snapshot ts .

```

theory Sensors
  imports Traffic Views
begin

locale sensors = traffic + view +
  fixes sensors::(cars) ⇒ traffic ⇒ (cars) ⇒ real
  assumes sensors-ge:(sensors e ts c) > 0
begin

definition space :: traffic ⇒ view ⇒ cars ⇒ real-int
  where space ts v c ≡ Abs-real-int (pos ts c, pos ts c + sensors (own v) ts c)

lemma left-space: left (space ts v c) = pos ts c
proof -
  have 1:pos ts c < pos ts c + sensors (own v) ts c using sensors-ge

```

```

    by (metis (no-types, opaque-lifting) less-add-same-cancel1 )
  show left (space ts v c) = pos ts c
    using space-def Abs-real-int-inverse 1 by simp
qed

lemma right-space: right (space ts v c) = pos ts c + sensors (own v) ts c
proof -
  have 1: pos ts c < pos ts c + sensors (own v) ts c using sensors-ge
    by (metis (no-types, opaque-lifting) less-add-same-cancel1 )
  show 3:right(space ts v c) = pos ts c + sensors (own v) ts c
    using space-def Abs-real-int-inverse 1 by simp
qed

lemma space-nonempty:left (space ts v c) < right (space ts v c)
  using left-space right-space sensors-ge by simp

end
end

```

9 Visible Length of Cars with Perfect Sensors

Given a sensor function, we can define the length of a car c as perceived by the owner of a view v . This length is restricted by the size of the extension of the view v , but always given by a continuous interval, which may possibly be degenerate (i.e., a point-interval).

The lemmas connect the end-points of the perceived length with the end-points of the current view. Furthermore, they show how the chopping and subview relations affect the perceived length of a car.

```

theory Length
  imports Sensors
begin

```

```

context sensors
begin

```

```

definition len:: view  $\Rightarrow$  traffic  $\Rightarrow$  cars  $\Rightarrow$  real-int
  where len-def :len v ( ts ) c ==
    if (left (space ts v c) > right (ext v))
      then Abs-real-int (right (ext v),right (ext v))
    else
      if (right (space ts v c) < left (ext v))
        then Abs-real-int (left (ext v),left (ext v))
      else
        Abs-real-int (max (left (ext v)) (left (space ts v c)),
          min (right (ext v)) (right (space ts v c)))

```

lemma *len-left*: $\text{left } ((\text{len } v \text{ ts}) c) \geq \text{left } (\text{ext } v)$
using *Abs-real-int-inverse left-leq-right sensors.len-def sensors-axioms* **by** *auto*

lemma *len-right*: $\text{right } ((\text{len } v \text{ ts}) c) \leq \text{right } (\text{ext } v)$
using *Abs-real-int-inverse left-leq-right sensors.len-def sensors-axioms* **by** *auto*

lemma *len-sub-int*: $\text{len } v \text{ ts } c \leq \text{ext } v$
using *less-eq-real-int-def len-left len-right* **by** *blast*

lemma *len-space-left*:
 $\text{left } (\text{space } ts \ v \ c) \leq \text{right } (\text{ext } v) \longrightarrow \text{left } (\text{len } v \ \text{ts } \ c) \geq \text{left } (\text{space } ts \ v \ c)$

proof

assume *assm*: $\text{left } (\text{space } ts \ v \ c) \leq \text{right } (\text{ext } v)$

then show $\text{left } (\text{len } v \ \text{ts } \ c) \geq \text{left } (\text{space } ts \ v \ c)$

proof (*cases* $\text{right } ((\text{space } ts \ v) \ c) < \text{left } (\text{ext } v)$)

case *True*

then show *?thesis* **using** *len-def len-left real-int.left-leq-right*

by (*meson le-less-trans not-less order.asym*)

next

case *False*

then have $\text{len } v \ \text{ts } \ c =$

$\text{Abs-real-int } ((\text{max } (\text{left } (\text{ext } v)) (\text{left } ((\text{space } ts \ v) \ c))),$
 $\text{min } (\text{right } (\text{ext } v)) (\text{right } ((\text{space } ts \ v) \ c)))$

using *len-def assm* **by** *auto*

then have $\text{left } (\text{len } v \ \text{ts } \ c) = \text{max } (\text{left } (\text{ext } v)) (\text{left } ((\text{space } ts \ v) \ c))$

using *Abs-real-int-inverse False assm real-int.left-leq-right*

by *auto*

then show *?thesis* **by** *linarith*

qed

qed

lemma *len-space-right*:

$\text{right } (\text{space } ts \ v \ c) \geq \text{left } (\text{ext } v) \longrightarrow \text{right } (\text{len } v \ \text{ts } \ c) \leq \text{right } (\text{space } ts \ v \ c)$

proof

assume *assm*: $\text{right } (\text{space } ts \ v \ c) \geq \text{left } (\text{ext } v)$

then show $\text{right } (\text{len } v \ \text{ts } \ c) \leq \text{right } (\text{space } ts \ v \ c)$

proof (*cases* $\text{left } ((\text{space } ts \ v) \ c) > \text{right } (\text{ext } v)$)

case *True*

then show *?thesis* **using** *len-def len-right real-int.left-leq-right*

by (*meson le-less-trans not-less order.asym*)

next

case *False*

then have $\text{len } v \ \text{ts } \ c =$

$\text{Abs-real-int } (\text{max } (\text{left } (\text{ext } v)) (\text{left } ((\text{space } ts \ v) \ c)),$
 $\text{min } (\text{right } (\text{ext } v)) (\text{right } ((\text{space } ts \ v) \ c)))$

using *len-def assm* **by** *auto*

then have $\text{right } (\text{len } v \ \text{ts } \ c) = \text{min } (\text{right } (\text{ext } v)) (\text{right } ((\text{space } ts \ v) \ c))$

using *Abs-real-int-inverse False assm real-int.left-leq-right*

by *auto*

then show *?thesis* by *linarith*
 qed
 qed

lemma *len-hchop-left-right-border*:

$(len\ v\ ts\ c = ext\ v) \wedge (v=v1\ \|\ v2) \longrightarrow (right\ (len\ v1\ ts\ c) = right\ (ext\ v1))$

proof

assume *asm*: $((len\ v\ ts)\ c = ext\ v) \wedge (v=v1\ \|\ v2)$

have *l1*: $left\ ((len\ v\ ts)\ c) = left\ (ext\ v)$ using *asm* by *auto*

from *asm* have *l2*: $left\ (ext\ v) = left\ (ext\ v1)$

by (*simp add: hchop-def real-int.rchop-def*)

from *l1* and *l2* have *l3*: $left\ ((len\ v\ ts)\ c) = left\ (ext\ v1)$ by *simp*

have *r1*: $right\ ((len\ v\ ts)\ c) = right\ (ext\ v)$ using *asm* by *auto*

have *r2*: $right\ (ext\ v1) \leq right\ (ext\ v)$

by (*metis (no-types, lifting) asm hchop-def real-int.rchop-def real-int.left-leq-right*)

have *r3*: $right\ ((len\ v1\ ts)\ c) \leq right\ (ext\ v1)$

using *len-right* by *blast*

show $right\ ((len\ v1\ ts)\ c) = right\ (ext\ v1)$

proof (*rule ccontr*)

assume *contra*: $right\ ((len\ v1\ ts)\ c) \neq right\ (ext\ v1)$

with *r3* have *less*: $right\ ((len\ v1\ ts)\ c) < right\ (ext\ v1)$ by *simp*

show *False*

proof (*cases left ((space ts v) c) ≤ right (ext v1)*)

assume *neg1*: $\neg left\ ((space\ ts\ v)\ c) \leq right\ (ext\ v1)$

have $right\ ((len\ v1\ ts)\ c) = right\ (ext\ v1)$

using *Abs-real-int-inverse left-space len-def neg1 right.rep-eq* by *auto*

with *contra* show *False* ..

next

assume *less1*: $left\ ((space\ ts\ v)\ c) \leq right\ (ext\ v1)$

show *False*

proof (*cases right ((space ts v) c) ≥ left (ext v1)*)

assume *neg2*: $\neg left\ (ext\ v1) \leq right\ ((space\ ts\ v)\ c)$

have $right\ ((len\ v1\ ts)\ c) = right\ (ext\ v1)$

proof –

have $(len\ v1\ ts)\ c = Abs-real-int\ (left\ (ext\ v1), left\ (ext\ v1))$

using *len-def neg2 asm hchop-def real-int.left-leq-right less1 space-def*

by *auto*

hence $right\ ((len\ v1\ ts)\ c) = left\ ((len\ v1\ ts)\ c)$

using *l3 asm contra less1 len-def neg2 r2 r3 real-int.left-leq-right*

by *auto*

with *l1* have *r4*: $right\ ((len\ v1\ ts)\ c) = right\ (ext\ v)$

using *asm l2 len-def neg2 asm hchop-def less1 real-int.left-leq-right r2 space-def*

by *auto*

hence $right\ (ext\ v) = right\ (ext\ v1)$

using *r2 r3* by *auto*

thus $right\ ((len\ v1\ ts)\ c) = right\ (ext\ v1)$

```

    using r4 by auto
  qed
  with contra show False ..
next
assume less2:left (ext v1) ≤ right ((space ts v) c)
have len-in-type:
  (max (left (ext v1)) (left ((space ts v) c)),
   min (right (ext v1)) (right ((space ts v) c)))
  ∈ {r :: real*real . fst r ≤ snd r}
  using Rep-real-int less1 less2 by auto
from less1 and less2 have len-def-v1:len v1 (ts) c =
  Abs-real-int ((max (left (ext v1)) (left ((space ts v) c))),
               min (right (ext v1)) (right ((space ts v) c)))
  using len-def assm hchop-def space-def by auto
with less have
  min (right (ext v1)) (right ((space ts v) c)) = right ((space ts v) c)
  using Abs-real-int-inverse len-in-type snd-conv by auto
hence right ((space ts v) c) ≤ right (ext v1) by simp
hence right ((space ts v) c) ≤ right (ext v)
  using r2 by linarith
from len-def-v1 and less and len-in-type
have right ((space ts v) c) < right (ext v1)
  using Abs-real-int-inverse sndI by auto
hence r4:right ((space ts v) c) < right (ext v)
  using r2 by linarith
from assm have len-v-in-type:
  (max (left (ext v)) (left ((space ts v) c)),
   min (right (ext v)) (right ((space ts v) c)))
  ∈ {r :: real*real . fst r ≤ snd r}
  using r4 l2 len-in-type by auto
hence right (len v (ts) c) ≠ right (ext v)
  using Abs-real-int-inverse Pair-inject r4 len-def real-int.left-leq-right
  surjective-pairing by auto
with r1 show False by best
qed
qed
qed
qed

```

lemma *len-hchop-left-left-border*:

$((len v ts) c = ext v) \wedge (v=v1||v2) \longrightarrow (left ((len v1 ts) c) = left (ext v1))$

proof

assume *assm*: $((len v ts) c = ext v) \wedge (v=v1||v2)$

have *l1*: $left ((len v ts) c) = left (ext v)$ **using** *assm* **by** *auto*

from *assm* **have** *l2*: $left (ext v) = left (ext v1)$

by (*simp add: hchop-def real-int.rchop-def*)

from *l1* **and** *l2* **have** *l3*: $left ((len v ts) c) = left (ext v1)$ **by** *simp*

have *r1*: $right ((len v ts) c) = right (ext v)$ **using** *assm* **by** *auto*

have *r2*: $right (ext v1) \leq right (ext v)$

```

  by (metis (no-types, lifting) assm hchop-def real-int.rchop-def
      real-int.left-leq-right )
have r3:right ((len v1 ts) c) ≤ right (ext v1)
  using len-right by blast
show (left ((len v1 ts) c) = left (ext v1))
proof (cases
  left ((space ts v) c) ≤ right (ext v1) ∧ right ((space ts v) c) ≥ left (ext v1))
case True
show (left ((len v1 ts) c) = left (ext v1))
proof (rule ccontr)
  assume neq: left (len v1 ( ts) c) ≠ left (ext v1)
  then have greater: left (len v1 ( ts) c) > left (ext v1)
    by (meson dual-order.order-iff-strict len-left)
  have len-in-type:
    (max (left (ext v1)) (left ((space ts v) c)),
     min (right (ext v1)) (right ((space ts v) c)))
    ∈ {r :: real*real . fst r ≤ snd r}
  using Rep-real-int True by auto
from True have len v1 ( ts) c =
  Abs-real-int ((max (left (ext v1)) (left ((space ts v) c)),
    min (right (ext v1)) (right ((space ts v) c)))
  using len-def assm hchop-def space-def by auto
hence maximum:
  left (len v1 ( ts) c) = max (left (ext v1)) (left ((space ts v) c))
  using Abs-real-int-inverse len-in-type by auto
have max (left (ext v1)) (left ((space ts v) c)) = left ((space ts v) c)
  using maximum neq by linarith
hence left ((space ts v) c) > left (ext v1)
  using greater maximum by auto
hence l4:left ((space ts v) c) > left (ext v) using l2 by auto
with assm have len-v-in-type:
  (max (left (ext v)) (left ((space ts v) c)),
   min (right (ext v)) (right ((space ts v) c)))
  ∈ {r :: real*real . fst r ≤ snd r}
  using len-in-type r2 by auto
hence left (len v ( ts) c) ≠ left (ext v)
  using Abs-real-int-inverse l4 sensors.len-def sensors-axioms by auto
thus False using l1 by best
qed
next
case False
then have
  ¬left ((space ts v) c) ≤ right (ext v1) ∨ ¬right ((space ts v) c) ≥ left (ext v1)
  by auto
then show (left ((len v1 ts) c) = left (ext v1))
proof
  assume negative: ¬ left ((space ts v) c) ≤ right (ext v1)
  then have len v1 ( ts) c = Abs-real-int (right (ext v1), right (ext v1))
    using len-def assm hchop-def space-def by auto

```

```

hence empty:left (len v1 ( ts) c) = right (len v1 ( ts) c)
by (metis real-int.chop-assoc2 real-int.chop-singleton-right real-int.rchop-def)
have len-geq:left(len v1 ( ts) c) ≥ left (ext v)
  using l2 len-left by auto
show left (len v1 ( ts) c) = left (ext v1)
proof (rule ccontr)
  assume contra:left (len v1 ( ts) c) ≠ left (ext v1)
  with len-left have left (ext v1) < left (len v1 ( ts) c)
    using dual-order.order-iff-strict by blast
  hence l5:left (ext v) < left (len v1 ( ts) c) using l2 by auto
  hence l6:left (len v ( ts) c) < left (len v1 ( ts) c) using l1 by auto
  show False
proof (cases left ((space ts v) c) ≤ right (ext v))
  case True
  have well-sp:left ((space ts v) c) ≤ right ((space ts v) c)
    using real-int.left-leq-right by auto
  have well-v:left (ext v) ≤ right (ext v)
    using real-int.left-leq-right by auto
  hence rs-geq-vl:right ((space ts v) c) ≥ left (ext v)
    using empty len-geq negative r3 well-sp by linarith
  from True and rs-geq-vl have len-in-type:
    (max (left (ext v)) (left ((space ts v) c)),
     min (right (ext v)) (right ((space ts v) c)))
    ∈ {r :: real*real . fst r ≤ snd r}
    using CollectD CollectI Rep-real-int fst-conv snd-conv by auto
  have len v (ts) c =
    Abs-real-int (max (left (ext v)) (left ((space ts v) c)),
                  min (right (ext v)) (right ((space ts v) c)))
    using len-def using True rs-geq-vl by auto
  hence max-less:
    max (left (ext v)) (left ((space ts v) c)) < left (len v1 ( ts) c)
    using Abs-real-int-inverse
    by (metis (full-types) l5 assm fst-conv left.rep-eq len-in-type)
  show False
    using empty max-less negative r3 by auto
next
  case False
  then have len v ( ts) c = Abs-real-int (right (ext v), right (ext v))
    using len-def by auto
  hence empty-len-v:left (len v ( ts) c) = right (ext v) using Abs-real-int-inverse

  by simp
  show False
    using l6 empty empty-len-v r2 r3 by linarith
qed
qed
next
have space ts v1 c ≤ space ts v c using assm hchop-def space-def by auto
hence r4:right (space ts v1 c) ≤ right (space ts v c)

```

using *less-eq-real-int-def* **by** *auto*
assume *left-outside*: \neg *left* (*ext v1*) \leq *right* ((*space ts v*) *c*)
hence *left* (*ext v1*) $>$ *right* (*space ts v1 c*) **using** *r4* **by** *linarith*
then have *len v1* (*ts*) *c* = *Abs-real-int* (*left* (*ext v1*),*left* (*ext v1*))
using *len-def* *assm* *hchop-def* *real-int.left-leq-right* *r1* *r2* *l1* *l2* *l3* *r3*
by (*meson* *le-less-trans* *less-trans* *not-less*)
thus (*left* ((*len v1 ts*) *c*) = *left* (*ext v1*))
using *Abs-real-int-inverse* **by** *auto*
qed
qed
qed

lemma *len-view-hchop-left*:
((*len v ts*) *c* = *ext v*) \wedge (*v=v1* \parallel *v2*) \longrightarrow ((*len v1 ts*) *c* = *ext v1*)
by (*metis* *Rep-real-int-inverse* *left.rep-eq* *len-hchop-left-left-border*
len-hchop-left-right-border *prod.collapse* *right.rep-eq*)

lemma *len-hchop-right-left-border*:
((*len v ts*) *c* = *ext v*) \wedge (*v=v1* \parallel *v2*) \longrightarrow (*left* ((*len v2 ts*) *c*) = *left* (*ext v2*))
proof
assume *assm*:((*len v ts*) *c* = *ext v*) \wedge (*v=v1* \parallel *v2*)
have *r1*:*right* ((*len v ts*) *c*) = *right* (*ext v*) **using** *assm* **by** *auto*
from *assm* **have** *r2*:*right* (*ext v*) = *right* (*ext v2*)
by (*simp* *add*: *hchop-def* *real-int.rchop-def*)
from *r1* **and** *r2* **have** *r3*:*right* ((*len v ts*) *c*) = *right* (*ext v2*) **by** *simp*
have *l1*:*left* ((*len v ts*) *c*) = *left* (*ext v*) **using** *assm* **by** *auto*
have *l2*:*left* (*ext v2*) \geq *left* (*ext v*)
using *assm* *less-eq-real-int-def* *real-int.chop-leq2* *view.hchop-def* **by** *blast*
have *l3*:*left* ((*len v2 ts*) *c*) \geq *left* (*ext v2*)
using *len-left* **by** *blast*
show *left* ((*len v2 ts*) *c*) = *left* (*ext v2*)
proof (*rule* *ccontr*)
assume *contra*:*left* ((*len v2 ts*) *c*) \neq *left* (*ext v2*)
with *l3* **have** *less*:*left* ((*len v2 ts*) *c*) $>$ *left* (*ext v2*) **by** *simp*
show *False*
proof (*cases* *left* ((*space ts v*) *c*) \leq *right* (*ext v2*))
assume *neg1*: \neg *left* ((*space ts v*) *c*) \leq *right* (*ext v2*)
have *left* ((*len v2 ts*) *c*) = *left* (*ext v2*)
proof –
have (*len v2 ts*) *c* = *Abs-real-int* (*right* (*ext v2*),*right* (*ext v2*))
using *len-def* *neg1* *assm* *hchop-def* *space-def* **by** *auto*
thus *left* ((*len v2 ts*) *c*) = *left* (*ext v2*)
using *assm* *l2* *l3* *len-def* *neg1* *r3* **by** *auto*
qed
with *contra* **show** *False* ..
next
assume *less1*:*left* ((*space ts v*) *c*) \leq *right* (*ext v2*)
show *False*
proof (*cases* *right* ((*space ts v*) *c*) \geq *left* (*ext v2*))

```

assume neg2:¬ left (ext v2) ≤ right ((space ts v) c)
have space ts v2 c ≤ space ts v c using assm hchop-def space-def by auto
hence right (space ts v2 c) ≤ right (space ts v c) using less-eq-real-int-def
by auto
with neg2 have greater:left (ext v2) > right (space ts v2 c) by auto
have left ((len v2 ts) c) = left (ext v2)
proof –
  have len-empty:(len v2 ts) c = Abs-real-int (left (ext v2),left (ext v2))
    using len-def neg2 assm hchop-def less1 space-def by auto
  have l4:left((len v2 ts)c) = left (ext v)
    using Abs-real-int-inverse len-def less neg2 assm hchop-def
    CollectI len-empty prod.collapse prod.inject by auto
  hence left (ext v) = left (ext v2)
    using l2 l3 by auto
  thus left ((len v2 ts) c) = left (ext v2) using l4 by auto
qed
with contra show False ..
next
assume less2:left (ext v2) ≤ right ((space ts v) c)
have len-in-type:
  (max (left (ext v2)) (left ((space ts v) c)),
   min (right (ext v2)) (right ((space ts v) c)))
  ∈ {r :: real*real . fst r ≤ snd r}
  using Rep-real-int less1 less2 by auto
from less1 and less2 have len-def-v2:len v2 ( ts) c =
  Abs-real-int (max (left (ext v2)) (left ((space ts v) c)),
               min (right (ext v2)) (right ((space ts v) c)))
  using len-def assm hchop-def space-def by auto
with less have
  max (left (ext v2)) (left ((space ts v) c)) = left ((space ts v) c)
  using Abs-real-int-inverse len-in-type snd-conv by auto
hence left ((space ts v) c) ≥ left (ext v2) by simp
hence left ((space ts v) c) ≥ left (ext v)
  using l2 by auto
from len-def-v2 and less and len-in-type
have left ((space ts v) c) > left (ext v2)
  using Abs-real-int-inverse sndI by auto
hence l5:left ((space ts v) c) > left (ext v)
  using l2 by linarith
with assm have len-v-in-type:
  (max (left (ext v)) (left (space ts v c)),
   min (right (ext v)) (right (space ts v c)))
  ∈ {r :: real*real . fst r ≤ snd r}
  using r2 len-in-type by auto
hence left (len v ( ts) c) ≠ left (ext v)
  using Abs-real-int-inverse Pair-inject l5 len-def real-int.left-leq-right
  surjective-pairing by auto
with l1 show False by best
qed

```

qed
 qed
 qed

lemma *len-hchop-right-right-border*:

$((len\ v\ ts)\ c = ext\ v) \wedge (v=v1\ \|\ v2) \longrightarrow (right\ ((len\ v2\ ts)\ c) = right\ (ext\ v2))$

proof

assume *assm*: $((len\ v\ ts)\ c = ext\ v) \wedge (v=v1\ \|\ v2)$

have *r1*: $right\ ((len\ v\ ts)\ c) = right\ (ext\ v)$ **using** *assm* **by** *auto*

from *assm* **have** *r2*: $right\ (ext\ v) = right\ (ext\ v2)$

by (*simp* *add*: *hchop-def* *real-int.rchop-def*)

from *r1* **and** *r2* **have** *r3*: $right\ ((len\ v\ ts)\ c) = right\ (ext\ v2)$ **by** *simp*

have *l1*: $left\ ((len\ v\ ts)\ c) = left\ (ext\ v)$ **using** *assm* **by** *auto*

have *l2*: $left\ (ext\ v2) \leq right\ (ext\ v)$

using *assm* *view.h-chop-middle2* **by** *blast*

have *l3*: $left\ ((len\ v2\ ts)\ c) \geq left\ (ext\ v2)$

using *len-left* **by** *blast*

show $right\ ((len\ v2\ ts)\ c) = right\ (ext\ v2)$

proof (*cases*

$left\ ((space\ ts\ v)\ c) \leq right\ (ext\ v2) \wedge right\ ((space\ ts\ v)\ c) \geq left\ (ext\ v2)$)

case *True*

show $right\ ((len\ v2\ ts)\ c) = right\ (ext\ v2)$

proof (*rule* *ccontr*)

assume *neg*: $right\ (len\ v2\ (ts)\ c) \neq right\ (ext\ v2)$

then **have** *lesser*: $right\ (len\ v2\ (ts)\ c) < right\ (ext\ v2)$

using *len-right-less-eq-real-def* **by** *blast*

have *len-in-type*:

$(max\ (left\ (ext\ v2))\ (left\ (space\ ts\ v\ c)),$
 $min\ (right\ (ext\ v2))\ (right\ (space\ ts\ v\ c)))$
 $\in \{r :: real*real . fst\ r \leq snd\ r\}$

using *Rep-real-int* *True* **by** *auto*

from *True* **have**

$len\ v2\ (ts)\ c =$

$Abs-real-int\ (max\ (left\ (ext\ v2))\ (left\ (space\ ts\ v\ c)),$
 $min\ (right\ (ext\ v2))\ (right\ (space\ ts\ v\ c)))$

using *len-def* *assm* *hchop-def* *space-def* **by** *auto*

hence *maximum*:

$right\ (len\ v2\ (ts)\ c) = min\ (right\ (ext\ v2))\ (right\ ((space\ ts\ v)\ c))$

using *Abs-real-int-inverse* *len-in-type* **by** *auto*

have *min-right*:

$min\ (right\ (ext\ v2))\ (right\ ((space\ ts\ v)\ c)) = right\ ((space\ ts\ v)\ c)$

using *maximum* *neg* **by** *linarith*

hence $right\ ((space\ ts\ v)\ c) < right\ (ext\ v2)$

using *lesser* *maximum* **by** *auto*

hence *right-v*: $right\ ((space\ ts\ v)\ c) < right\ (ext\ v)$

using *r2* **by** *auto*

have *right-inside*: $right\ ((space\ ts\ v)\ c) \geq left\ (ext\ v)$

by (*meson* *True* *assm* *less-eq-real-int-def* *less-eq-view-ext-def*
order-trans *view.horizontal-chop-leq2*)

```

with assm and True and right-inside
have len-v-in-type:
  (max (left (ext v)) (left (space ts v c)),
   min (right (ext v)) (right (space ts v c)))
  ∈ {r :: real*real . fst r ≤ snd r}
  using min-right r2 real-int.left-leq-right by auto
hence right (len v (ts) c) ≠ right (ext v)
  using Abs-real-int-inverse Pair-inject right-v len-def
  real-int.left-leq-right surjective-pairing
  by auto
thus False using r1 by best
qed
next
case False
then have ¬left ((space ts v) c) ≤ right (ext v2) ∨
  ¬right ((space ts v) c) ≥ left (ext v2)
  by auto
thus right ((len v2 ts) c) = right (ext v2)
proof
  assume negative: ¬ left ((space ts v) c) ≤ right (ext v2)
  show ?thesis
    using left-space negative r1 r3 sensors.len-def sensors-axioms by auto
next
  assume left-outside: ¬ left (ext v2) ≤ right ((space ts v) c)
  hence left (ext v2) > right (space ts v2 c)
    using assm hchop-def space-def by auto
  then have len: len v2 (ts) c = Abs-real-int (left (ext v2), left (ext v2))
  by (metis (no-types, opaque-lifting) len-def l2 le-less-trans not-less order.assym
    space-nonempty r2)
  show (right ((len v2 ts) c) = right (ext v2))
proof (cases right ((space ts v) c) ≥ left (ext v))
  assume ¬ left (ext v) ≤ right ((space ts v) c)
  hence len-empty: len v (ts) c = Abs-real-int (left (ext v), left (ext v))
    using len-def real-int.left-leq-right Abs-real-int-inverse
    by (meson less-trans not-less space-nonempty)
  show (right ((len v2 ts) c) = right (ext v2))
  by (metis (no-types, opaque-lifting) Rep-real-int-inverse assm dual-order.antisym
    left.rep-eq len len-empty prod.collapse real-int.chop-singleton-left
    real-int.rchop-def right.rep-eq view.h-chop-middle1 view.hchop-def)
next
  assume left (ext v) ≤ right ((space ts v) c)
  then show ?thesis
    using l2 left-outside len-space-right r1 by fastforce
qed
qed
qed
qed

```


lemma *len-view-hchop-right*:

$((len\ v\ ts)\ c = ext\ v) \wedge (v=v1\ \|\ v2) \longrightarrow ((len\ v2\ ts)\ c = ext\ v2)$
by (*metis Rep-real-int-inverse left.rep-eq len-hchop-right-left-border len-hchop-right-right-border prod.collapse right.rep-eq*)

lemma *len-compose-hchop*:

$(v=v1\ \|\ v2) \wedge (len\ v1\ (ts)\ c = ext\ v1) \wedge (len\ v2\ (ts)\ c = ext\ v2)$
 $\longrightarrow (len\ v\ (ts)\ c = ext\ v)$

proof

assume *assm*: $(v=v1\ \|\ v2) \wedge (len\ v1\ (ts)\ c = ext\ v1) \wedge (len\ v2\ (ts)\ c = ext\ v2)$

then have *left-v1:left*: $left\ (len\ v1\ (ts)\ c) = left\ (ext\ v1)$ **by** *auto*

from *assm* **have** *right-v1:right*: $right\ (len\ v1\ (ts)\ c) = left\ (ext\ v2)$

by (*simp add: hchop-def real-int.rchop-def*)

from *assm* **have** *left-v2:left*: $left\ (len\ v2\ (ts)\ c) = right\ (ext\ v1)$

using *right-v1* **by** *auto*

from *assm* **have** *right-v2:right*: $right\ (len\ v2\ (ts)\ c) = right\ (ext\ v2)$ **by** *auto*

show $(len\ v\ (ts)\ c = ext\ v)$

proof (*cases left ((space ts v) c) > right (ext v)*)

case *True*

then have $left\ (space\ ts\ v\ c) > right\ (ext\ v2)$ **using** *assm right-v2*

by (*simp add: hchop-def real-int.rchop-def*)

then have $left\ (space\ ts\ v2\ c) > right\ (ext\ v2)$

using *assm hchop-def sensors.space-def sensors-axioms* **by** *auto*

then have $len\ v2\ ts\ c = Abs-real-int(right\ (ext\ v2), right\ (ext\ v2))$

using *len-def* **by** *simp*

then have $ext\ v2 = Abs-real-int(right\ (ext\ v2), right\ (ext\ v2))$ **using** *assm* **by**

simp

then have $\|ext\ v2\| = 0$

by (*metis Rep-real-int-inverse fst-conv left.rep-eq*

real-int.chop-singleton-right real-int.length-zero-iff-borders-eq

real-int.rchop-def right.rep-eq snd-conv surj-pair)

then have $ext\ v = ext\ v1$

using *assm hchop-def real-int.rchop-def real-int.chop-empty2*

by *simp*

then show *?thesis*

using *assm hchop-def len-def sensors.space-def sensors-axioms*

by *auto*

next

case *False*

then have *in-left:left*: $left\ (space\ ts\ v\ c) \leq right\ (ext\ v)$ **by** *simp*

show $len\ v\ ts\ c = ext\ v$

proof (*cases right (space ts v) c < left (ext v)*)

case *True*

then have $right\ (space\ ts\ v\ c) < left\ (ext\ v1)$ **using** *assm left-v1*

by (*simp add: hchop-def real-int.rchop-def*)

then have *out-v1:right*: $right\ (space\ ts\ v1\ c) < left\ (ext\ v1)$

using *assm hchop-def sensors.space-def sensors-axioms* **by** *auto*

then have $len\ v1\ ts\ c = Abs-real-int(left\ (ext\ v1), left\ (ext\ v1))$

using *len-def in-left*

```

    by (meson le-less-trans less-trans not-le real-int.left-leq-right)
  then have ext v1 = Abs-real-int (left (ext v1), left (ext v1)) using assm by
simp
  then have ||ext v1|| = 0
    by (metis add.right-neutral real-int.chop-singleton-left
      real-int.length-zero-iff-borders-eq real-int.rchop-def real-int.shift-def
      real-int.shift-zero)
  then have ext v = ext v2 using assm hchop-def real-int.rchop-def real-int.chop-empty1
    by auto
  then show ?thesis
    using assm hchop-def len-def sensors.space-def sensors-axioms by auto
next
case False
then have in-right:right (space ts v c) ≥ left (ext v) by simp
have f1: own v = own v2 using assm hchop-def
  by (auto)
have f2: own v = own v1
  using assm hchop-def by auto
have chop:R-Chop(ext v,ext v1,ext v2) using assm hchop-def
  by (auto)
have len:len v ts c = Abs-real-int(max (left (ext v)) (left (space ts v c)),
  min (right (ext v)) (right (space ts v c)))
  using len-def in-left in-right by simp
have len1:len v1 ts c = Abs-real-int(max (left (ext v1)) (left (space ts v1 c)),
  min (right (ext v1)) (right (space ts v1 c)))
  by (metis assm f2 f1 chop assm in-left in-right len-def len-space-left
    not-le real-int.rchop-def space-def)
then have max (left (ext v1)) (left (space ts v1 c)) = left (len v1 ts c)
  by (metis assm chop f1 f2 in-left len-space-left max.orderE
    real-int.rchop-def space-def)
then have left-border:max (left (ext v1)) (left (space ts v1 c)) = left (ext v1)
  using left-v1 by simp
have len2:len v2 ts c = Abs-real-int(max (left (ext v2)) (left (space ts v2 c)),
  min (right (ext v2)) (right (space ts v2 c)))
  by (metis len-def in-left in-right assm f2 f1 chop len-space-right not-le
    real-int.rchop-def space-def)
then have min (right (ext v2)) (right (space ts v2 c)) = right (len v2 ts c)
  by (metis assm chop f1 f2 in-right len-space-right min.absorb-iff1
    real-int.rchop-def space-def)
then have right-border:
  min (right (ext v2)) (right (space ts v2 c)) = right (ext v2)
  using right-v2 by simp
have left (space ts v c) = left (space ts v1 c)
  using assm hchop-def sensors.space-def sensors-axioms by auto
then have max:
  max (left (ext v)) (left (space ts v c))
  = max (left (ext v1)) (left (space ts v1 c))
  using assm hchop-def real-int.rchop-def by auto
have right (space ts v c) = right (space ts v2 c)

```

```

    using assm hchop-def sensors.space-def sensors-axioms by auto
  then have min:
    min (right (ext v)) (right (space ts v c))
    = min (right (ext v2)) (right (space ts v2 c))
  using assm hchop-def real-int.rchop-def by auto
  show ?thesis
  by (metis min max left-border right-border False add.right-neutral
    chop in-left len-def not-le real-int.rchop-def real-int.shift-def
    real-int.shift-zero)
qed
qed
qed

```

lemma *len-stable*: $(v=v1 \dashv\vdash v2) \longrightarrow \text{len } v1 \text{ ts } c = \text{len } v2 \text{ ts } c$

proof

```

  assume assm:v=v1--v2
  then have ext-eq1:ext v = ext v1 and ext-eq2:ext v = ext v2
  using vchop-def by auto
  hence ext1-eq-ext2:ext v1 = ext v2 by simp
  show len v1 ts c = len v2 ts c
  using assm ext1-eq-ext2 left-space right-space sensors.len-def sensors-axioms
    view.vertical-chop-own-trans by auto

```

qed

lemma *len-empty-on-subview1*:

$\| \text{len } v \text{ (ts) } c \| = 0 \wedge (v=v1 \| v2) \longrightarrow \| \text{len } v1 \text{ (ts) } c \| = 0$

proof

```

  assume assm: $\| \text{len } v \text{ ( ts ) } c \| = 0 \wedge (v=v1 \| v2)$ 
  then have len-v-borders:left (len v ( ts ) c) = right (len v ( ts ) c)
  by (simp add:real-int.length-zero-iff-borders-eq)
  show  $\| \text{len } v1 \text{ ( ts ) } c \| = 0$ 
  proof (cases left ((space ts v) c) > right (ext v1))
    assume left-outside-v1:left ((space ts v) c) > right (ext v1)
    thus  $\| \text{len } v1 \text{ ( ts ) } c \| = 0$ 
    using Abs-real-int-inverse assm fst-conv hchop-def len-def real-int.length-zero-iff-borders-eq
      mem-Collect-eq snd-conv space-def by auto
  next
    assume left-inside-v1: $\neg \text{left } ((\text{space } ts \ v) \ c) > \text{right } (\text{ext } v1)$ 
    show  $\| \text{len } v1 \text{ ( ts ) } c \| = 0$ 
    proof (cases left (ext v1) > right ((space ts v) c))
      assume right-outside-v1:left (ext v1) > right ((space ts v) c)
      hence left (ext v1) > right ((space ts v1) c) using assm hchop-def space-def
      by auto
      thus  $\| \text{len } v1 \text{ ( ts ) } c \| = 0$ 
      using assm hchop-def len-def real-int.length-def Abs-real-int-inverse by auto
    next
      assume right-inside-v1: $\neg \text{left } (\text{ext } v1) > \text{right } ((\text{space } ts \ v) \ c)$ 
      have len-v1:

```

$len\ v1\ (ts)\ c = Abs\text{-}real\text{-}int\ (max\ (left\ (ext\ v1))\ (left\ (space\ ts\ v\ c)),$
 $min\ (right\ (ext\ v1))\ (right\ (space\ ts\ v\ c)))$

using *left-inside-v1 len-def right-inside-v1 assm hchop-def space-def* **by** *auto*
from *left-inside-v1* **and** *right-inside-v1* **have** *inside-v*:
 $\neg left\ (space\ ts\ v\ c) > right\ (ext\ v) \wedge \neg left\ (ext\ v) > right\ (space\ ts\ v\ c)$

proof –
have *fst (Rep-real-int (ext v2)) ≤ snd (Rep-real-int (ext v))*
using *assm view.h-chop-middle2* **by** *force*
then show *?thesis*
using *assm left-inside-v1 real-int.rchop-def right-inside-v1 view.hchop-def*
by *force*

qed
hence *len-v*:
 $len\ v\ ts\ c = Abs\text{-}real\text{-}int\ (max\ (left\ (ext\ v))\ (left\ (space\ ts\ v\ c)),$
 $min\ (right\ (ext\ v))\ (right\ (space\ ts\ v\ c)))$

by (*simp add: len-def*)

have *less-eq*:
 $max\ (left\ (ext\ v))\ (left\ (space\ ts\ v\ c))$
 $\leq min\ (right\ (ext\ v))\ (right\ (space\ ts\ v\ c))$
using *inside-v real-int.left-leq-right* **by** *auto*

from *len-v* **have** *len-v-empty*:
 $max\ (left\ (ext\ v))\ (left\ ((space\ ts\ v)\ c))$
 $= min\ (right\ (ext\ v))\ (right\ ((space\ ts\ v)\ c))$
using *Abs-real-int-inverse Rep-real-int-inverse inside-v*
len-v-borders local.less-eq **by** *auto*

have *left-len-eq*:
 $max\ (left\ (ext\ v))\ (left\ (space\ ts\ v\ c))$
 $= max\ (left\ (ext\ v1))\ (left\ (space\ ts\ v\ c))$
using *assm hchop-def real-int.rchop-def* **by** *auto*

have *right-len-leq*:
 $min\ (right\ (ext\ v))\ (right\ (space\ ts\ v\ c))$
 $\geq min\ (right\ (ext\ v1))\ (right\ (space\ ts\ v\ c))$
by (*metis (no-types, opaque-lifting) assm min.bounded-iff min-less-iff-conj*)

not-le

$order\ refl\ real\text{-}int.rchop\text{-}def\ view.h\text{-}chop\text{-}middle2\ view.hchop\text{-}def$

hence *left-geq-right*:
 $max\ (left\ (ext\ v1))\ (left\ (space\ ts\ v\ c))$
 $\geq min\ (right\ (ext\ v1))\ (right\ (space\ ts\ v\ c))$
using *left-len-eq len-v-empty* **by** *auto*

thus $\|len\ v1\ (ts)\ c\| = 0$

proof –
have *f1*:
 $\neg max\ (left\ (ext\ v))\ (left\ (space\ ts\ v\ c))$
 $\leq min\ (right\ (ext\ v1))\ (right\ (space\ ts\ v\ c))$
 \vee
 $min\ (right\ (ext\ v1))\ (right\ (space\ ts\ v\ c))$
 $= max\ (left\ (ext\ v))\ (left\ (space\ ts\ v\ c))$
by (*metis antisym-conv left-geq-right left-len-eq*)

have

```

 $\wedge r. \neg \text{left } (\text{ext } v1) \leq r$ 
 $\vee \neg \text{left } (\text{space } ts \ v \ c) \leq r$ 
 $\vee \text{max } (\text{left } (\text{ext } v)) (\text{left } (\text{space } ts \ v \ c)) \leq r$ 
using left-len-eq by auto
then have
   $\text{min } (\text{right } (\text{ext } v1)) (\text{right } (\text{space } ts \ v \ c))$ 
   $= \text{max } (\text{left } (\text{ext } v)) (\text{left } (\text{space } ts \ v \ c))$ 
using f1 inside-v left-inside-v1 real-int.left-leq-right by force
then show ?thesis
using assm left-len-eq len-v len-v1 len-v-empty by auto
qed
qed
qed
qed

```

lemma *len-empty-on-subview2*:

$\| \text{len } v \ ts \ c \| = 0 \wedge (v=v1 \| v2) \longrightarrow \| \text{len } v2 \ ts \ c \| = 0$

proof

assume *assm*: $\| \text{len } v \ (\ ts) \ c \| = 0 \wedge (v=v1 \| v2)$

then have *len-v-borders*: $\text{left } (\text{len } v \ (\ ts) \ c) = \text{right } (\text{len } v \ (\ ts) \ c)$

by (*simp add:real-int.length-zero-iff-borders-eq*)

show $\| \text{len } v2 \ (\ ts) \ c \| = 0$

proof (*cases left ((space ts v) c) > right (ext v2)*)

assume *left-outside-v2*: $((\text{space } ts \ v) \ c) > \text{right } (\text{ext } v2)$

thus $\| \text{len } v2 \ (\ ts) \ c \| = 0$

using *Abs-real-int-inverse assm fst-conv hchop-def len-def*

real-int.length-zero-iff-borders-eq mem-Collect-eq snd-conv space-def

by *auto*

next

assume *left-inside-v2*: $\neg \text{left } (\text{space } ts \ v \ c) > \text{right } (\text{ext } v2)$

show $\| \text{len } v2 \ (\ ts) \ c \| = 0$

proof (*cases left (ext v2) > right (space ts v) c*)

assume *right-outside-v2*: $\text{left } (\text{ext } v2) > \text{right } ((\text{space } ts \ v) \ c)$

thus $\| \text{len } v2 \ (\ ts) \ c \| = 0$

using *Abs-real-int-inverse assm fst-conv hchop-def len-def*

real-int.length-zero-iff-borders-eq mem-Collect-eq snd-conv

right-outside-v2 space-def

by *auto*

next

assume *right-inside-v2*: $\neg \text{left } (\text{ext } v2) > \text{right } ((\text{space } ts \ v) \ c)$

have *len-v2*:

$\text{len } v2 \ ts \ c = \text{Abs-real-int } (\text{max } (\text{left } (\text{ext } v2)) (\text{left } (\text{space } ts \ v \ c)),$
 $\text{min } (\text{right } (\text{ext } v2)) (\text{right } (\text{space } ts \ v \ c)))$

using *left-inside-v2 len-def right-inside-v2 assm hchop-def space-def* **by** *auto*

from *left-inside-v2* **and** *right-inside-v2* **have** *inside-v*:

$\neg \text{left } ((\text{space } ts \ v) \ c) > \text{right } (\text{ext } v) \wedge \neg \text{left } (\text{ext } v) > \text{right } ((\text{space } ts \ v) \ c)$

proof –

have $\text{left } (\text{ext } v) \leq \text{right } (\text{ext } v1)$

using *assm view.h-chop-middle1* **by** *auto*

```

then show ?thesis
  using assm left-inside-v2 real-int.rchop-def right-inside-v2 view.hchop-def
  by force
qed
hence len-v:
  
$$\text{len } v \text{ ts } c = \text{Abs-real-int } (\max (\text{left } (\text{ext } v)) (\text{left } (\text{space } \text{ts } v \text{ c})),$$


$$\min (\text{right } (\text{ext } v)) (\text{right } (\text{space } \text{ts } v \text{ c})))$$

  by (simp add: len-def)
have less-eq:
  
$$\max (\text{left } (\text{ext } v)) (\text{left } (\text{space } \text{ts } v \text{ c}))$$


$$\leq \min (\text{right } (\text{ext } v)) (\text{right } (\text{space } \text{ts } v \text{ c}))$$

  using inside-v real-int.left-leq-right by auto
from len-v have len-v-empty:
  
$$\max (\text{left } (\text{ext } v)) (\text{left } (\text{space } \text{ts } v \text{ c}))$$


$$= \min (\text{right } (\text{ext } v)) (\text{right } (\text{space } \text{ts } v \text{ c}))$$

  using Abs-real-int-inverse Rep-real-int-inverse inside-v
  using len-v-borders local.less-eq by auto
have left-len-eq:
  
$$\max (\text{left } (\text{ext } v)) (\text{left } (\text{space } \text{ts } v \text{ c}))$$


$$\leq \max (\text{left } (\text{ext } v2)) (\text{left } (\text{space } \text{ts } v \text{ c}))$$

  by (metis (no-types, opaque-lifting) assm left-leq-right max.mono order-refl
real-int.rchop-def view.hchop-def)
have right-len-leq:
  
$$\min (\text{right } (\text{ext } v)) (\text{right } (\text{space } \text{ts } v \text{ c}))$$


$$= \min (\text{right } (\text{ext } v2)) (\text{right } (\text{space } \text{ts } v \text{ c}))$$

  using assm real-int.rchop-def view.hchop-def by auto
hence left-geq-right:
  
$$\max (\text{left } (\text{ext } v2)) (\text{left } (\text{space } \text{ts } v \text{ c}))$$


$$\geq \min (\text{right } (\text{ext } v2)) (\text{right } (\text{space } \text{ts } v \text{ c}))$$

  using left-len-eq len-v-empty by auto
then have
  
$$\max (\text{left } (\text{ext } v2)) (\text{left } (\text{space } \text{ts } v2 \text{ c}))$$


$$\geq \min (\text{right } (\text{ext } v2)) (\text{right } (\text{space } \text{ts } v2 \text{ c}))$$

  using assm hchop-def space-def by auto
then have
  
$$\max (\text{left } (\text{ext } v2)) (\text{left } (\text{space } \text{ts } v2 \text{ c}))$$


$$= \min (\text{right } (\text{ext } v2)) (\text{right } (\text{space } \text{ts } v2 \text{ c}))$$

  by (metis (no-types, opaque-lifting) antisym-conv assm hchop-def len-v-empty
max-def min.bounded-iff not-le space-def right-inside-v2 right-len-leq
view.h-chop-middle2)
thus  $\|\text{len } v2 \text{ ( ts) } c\| = 0$ 
  by (metis (no-types, opaque-lifting) assm hchop-def len-v len-v2 len-v-empty
space-def right-len-leq)
qed
qed
qed

```

lemma *len-hchop-add*:

$$(v=v1 \| v2) \longrightarrow \|\text{len } v \text{ ts } c\| = \|\text{len } v1 \text{ ts } c\| + \|\text{len } v2 \text{ ts } c\|$$

proof
assume *chop*: $v=v1 \parallel v2$
show $\|len\ v\ ts\ c\| = \|len\ v1\ ts\ c\| + \|len\ v2\ ts\ c\|$
proof (*cases left* $((space\ ts\ v)\ c) > right\ (ext\ v)$)
assume *outside-right:left* $(space\ ts\ v\ c) > right\ (ext\ v)$
hence *len-zero*: $\|len\ v\ (ts)\ c\| = 0$
by (*simp add: Abs-real-int-inverse len-def real-int.length-zero-iff-borders-eq snd-eqD*)
with chop have $\|len\ v1\ ts\ c\| + \|len\ v2\ ts\ c\| = 0$
by (*metis add-cancel-right-left len-empty-on-subview1 len-empty-on-subview2*)
thus *?thesis* **using** *len-zero* **by** (*simp*)
next
assume *inside-right*: $\neg left\ ((space\ ts\ v)\ c) > right\ (ext\ v)$
show $\|len\ v\ ts\ c\| = \|len\ v1\ ts\ c\| + \|len\ v2\ ts\ c\|$
proof (*cases left* $(ext\ v) > right\ ((space\ ts\ v)\ c)$)
assume *outside-left*: $left\ (ext\ v) > right\ (space\ ts\ v\ c)$
hence *len-zero*: $\|len\ v\ (ts)\ c\| = 0$
by (*simp add: Abs-real-int-inverse len-def real-int.length-zero-iff-borders-eq snd-eqD*)
with chop have $\|len\ v1\ ts\ c\| + \|len\ v2\ ts\ c\| = 0$
by (*metis add-cancel-right-left len-empty-on-subview1 len-empty-on-subview2*)
thus *?thesis* **using** *len-zero* **by** (*simp*)
next
assume *inside-left*: $\neg left\ (ext\ v) > right\ ((space\ ts\ v)\ c)$
hence *len-def-v*: $len\ v\ (ts)\ c =$
 $Abs-real-int\ ((max\ (left\ (ext\ v))\ (left\ ((space\ ts\ v)\ c))),$
 $min\ (right\ (ext\ v))\ (right\ ((space\ ts\ v)\ c)))$
using *len-def inside-left inside-right* **by** *simp*
from *inside-left and inside-right* **have**
 $len-in-type:(max\ (left\ (ext\ v))\ (left\ ((space\ ts\ v)\ c)),$
 $min\ (right\ (ext\ v))\ (right\ ((space\ ts\ v)\ c)))$
 $\in \{r :: real*real . fst\ r \leq\ snd\ r\}$
using *CollectD CollectI Rep-real-int fst-conv snd-conv* **by** *auto*
show $\|len\ v\ (ts)\ c\| = \|len\ v1\ (ts)\ c\| + \|len\ v2\ (ts)\ c\|$
proof (*cases right* $(len\ v\ (ts)\ c) < right\ (ext\ v1)$)
assume *inside-v1:right* $(len\ v\ (ts)\ c) < right\ (ext\ v1)$
then have *min-less-v1*:
 $min\ (right\ (ext\ v))\ (right\ (space\ ts\ v\ c)) < right\ (ext\ v1)$
using *Abs-real-int-inverse len-in-type len-def-v* **by** *auto*
hence *outside-v2:right* $((space\ ts\ v)\ c) < left\ (ext\ v2)$
proof –
have $left\ (ext\ v2) = right\ (ext\ v1)$
using *chop real-int.rchop-def view.hchop-def* **by** *force*
then show *?thesis*
by (*metis (no-types) chop dual-order.order-iff-strict*
 $min-less-iff-conj\ min-less-v1\ not-less\ view.h-chop-middle2$)
qed
hence *len-v2-0*: $\|len\ v2\ (ts)\ c\| = 0$ **using** *Abs-real-int-inverse len-def*
 $real-int.length-zero-iff-borders-eq\ outside-v2\ snd-eqD\ Rep-real-int-inverse$

```

    chop hchop-def prod.collapse real-int.rchop-def real-int.chop-singleton-right
      space-def
    by auto
  have inside-left-v1:  $\neg \text{left } (\text{ext } v1) > \text{right } ((\text{space } ts \ v) \ c)$ 
    using chop hchop-def inside-left real-int.rchop-def by auto
  have inside-right-v1:  $\neg \text{left } ((\text{space } ts \ v) \ c) > \text{right } (\text{ext } v1)$ 
    by (meson inside-right less-trans min-less-iff-disj min-less-v1
      order.asym space-nonempty)
  have len1-def:  $\text{len } v1 \ ( \ ts) \ c =$ 
    Abs-real-int ((max (left (ext v1)) (left ((space ts v) c))),
      min (right (ext v1)) (right ((space ts v) c)))
    using len-def inside-left-v1 inside-right-v1 chop hchop-def space-def
    by auto
  hence  $\| \text{len } v \ ts \ c \| = \| \text{len } v1 \ ts \ c \|$ 
  proof -
    have  $\text{right } (\text{ext } v1) \leq \text{right } (\text{ext } v2)$ 
      using chop left-leq-right real-int.rchop-def view.hchop-def by auto
    then show ?thesis
      using chop len1-def len-def-v min-less-v1 real-int.rchop-def view.hchop-def
      by auto
  qed
  thus  $\| \text{len } v \ ts \ c \| = \| \text{len } v1 \ ts \ c \| + \| \text{len } v2 \ ts \ c \|$ 
    using len-v2-0 by (simp)
next
  assume r-inside-v2:  $\neg \text{right } (\text{len } v \ ( \ ts) \ c) < \text{right } (\text{ext } v1)$ 
  show  $\| \text{len } v \ ( \ ts) \ c \| = \| \text{len } v1 \ ( \ ts) \ c \| + \| \text{len } v2 \ ( \ ts) \ c \|$ 
  proof (cases left (len v (ts) c) > left (ext v2))
    assume inside-v2:  $\text{left } (\text{len } v \ ( \ ts) \ c) > \text{left } (\text{ext } v2)$ 
    hence max-geq-v1:  $\text{max } (\text{left } (\text{ext } v)) \ (\text{left } ((\text{space } ts \ v) \ c)) > \text{left } (\text{ext } v2)$ 
      using Abs-real-int-inverse len-in-type len-def by (simp)
    hence outside-v1:  $\text{left } ((\text{space } ts \ v) \ c) > \text{right } (\text{ext } v1)$ 
    proof -
      have  $\text{left } (\text{ext } v) \leq \text{right } (\text{ext } v1)$ 
        by (meson chop view.h-chop-middle1)
      then show ?thesis
        using chop max-geq-v1 real-int.rchop-def view.hchop-def by fastforce
    qed
  hence len-v1-0:  $\| \text{len } v1 \ ts \ c \| = 0$ 
    using Abs-real-int-inverse len-def real-int.length-zero-iff-borders-eq
      outside-v1 snd-eqD Rep-real-int-inverse chop hchop-def prod.collapse
      real-int.rchop-def real-int.chop-singleton-right space-def
    by auto
  have inside-left-v2:  $\neg \text{left } (\text{ext } v2) > \text{right } ((\text{space } ts \ v) \ c)$ 
    by (meson inside-left less-max-iff-disj less-trans max-geq-v1 order.asym
      space-nonempty)
  have inside-right-v2:  $\neg \text{left } ((\text{space } ts \ v) \ c) > \text{right } (\text{ext } v2)$ 
    using chop hchop-def inside-right real-int.rchop-def by auto
  have len2-def:  $\text{len } v2 \ ( \ ts) \ c =$ 
    Abs-real-int ((max (left (ext v2)) (left ((space ts v) c))),

```



```

      min (right (ext v2)) (right ((space ts v) c)))
    using len-def inside-left-v2 inside-right-v2 hchop-def chop space-def
    by auto
  hence ||len v ts c|| = ||len v2 ts c||
  proof -
    have left (ext v) ≤ left (ext v2)
      by (metis (no-types) chop real-int.rchop-def view.h-chop-middle1
          view.hchop-def)
    then show ?thesis
      using chop inside-left inside-right len2-def len-def outside-v1
          real-int.rchop-def view.hchop-def
      by auto
  qed
  thus ||len v ts c|| = ||len v1 ts c|| + ||len v2 ts c||
    using len-v1-0 by (simp)
next
  assume l-inside-v1: ¬left (len v (ts) c) > left (ext v2)
  have inside-left-v1: ¬left (ext v1) > right ((space ts v) c)
    using chop hchop-def inside-left real-int.rchop-def by auto
  have inside-right-v1: ¬left ((space ts v) c) > right (ext v1)
    using Abs-real-int-inverse chop hchop-def l-inside-v1 len-in-type
        len-def real-int.rchop-def
    by auto
  hence len1-def: len v1 (ts) c =
    Abs-real-int ((max (left (ext v1)) (left ((space ts v) c))),
        min (right (ext v1)) (right ((space ts v) c)))
    using inside-left-v1 inside-right-v1 len-def chop hchop-def space-def
    by (simp)
  from inside-left-v1 and inside-right-v1 have len1-in-type:
    (max (left (ext v1)) (left (space ts v c)),
      min (right (ext v1)) (right (space ts v c)))
    ∈ {r :: real*real . fst r ≤ snd r}
    using CollectD CollectI Rep-real-int fst-conv snd-conv by auto
  have inside-left-v2: ¬left (ext v2) > right ((space ts v) c)
    using real-int.rchop-def hchop-def inside-left chop Abs-real-int-inverse
        len-def-v len-in-type r-inside-v2 snd-conv
    by auto
  have inside-right-v2: ¬left ((space ts v) c) > right (ext v2)
    using Abs-real-int-inverse chop hchop-def l-inside-v1 len-in-type len-def
        real-int.rchop-def
    by auto
  hence len2-def: len v2 ts c =
    Abs-real-int (max (left (ext v2)) (left (space ts v c)),
        min (right (ext v2)) (right (space ts v c)))
    using inside-left-v2 inside-right-v2 len-def chop hchop-def space-def
    by (auto)
  from inside-left-v2 and inside-right-v2 have len2-in-type:
    (max (left (ext v2)) (left (space ts v c)),
      min (right (ext v2)) (right (space ts v c)))

```

```

    ∈ {r :: real*real . fst r ≤ snd r}
    using CollectD CollectI Rep-real-int fst-conv snd-conv
    by auto
  have left-v-v1:left (ext v) = left (ext v1)
    using chop hchop-def real-int.rchop-def by auto
  have max:
    max (left (ext v)) (left (space ts v c)) =
    max (left (ext v1)) (left (space ts v c))
    using left-v-v1 by auto
  have right-v-v2:right (ext v) = right (ext v2)
    using chop hchop-def real-int.rchop-def by auto
  have min: (min (right (ext v)) (right ((space ts v) c))) =
    (min (right (ext v2)) (right ((space ts v) c)))
    using right-v-v2 by auto
  from max have left-len-v1-v:left (len v (ts) c) = left (len v1 (ts) c)
    using Abs-real-int-inverse fst-conv len1-def len1-in-type
    len-def-v len-in-type
    by auto
  from min have right-len-v2-v:right (len v (ts) c) = right (len v2 (ts) c)
    using Abs-real-int-inverse fst-conv len1-def len2-in-type len-def-v
    len-in-type using len2-def snd-eqD by auto
  have right (len v1 (ts) c) = left (len v2 (ts) c)
    using Abs-real-int-inverse chop hchop-def len1-def len1-in-type len2-def
    len2-in-type real-int.rchop-def
    by auto
  thus ||len v ts c|| = ||len v1 ts c|| + ||len v2 ts c||
    using left-len-v1-v real-int.consec-add right-len-v2-v by simp
qed
qed
qed
qed
qed

```

lemma *len-non-empty-inside*:

```

||len v (ts) c|| > 0
  → left (space ts v c) < right (ext v) ∧ right (space ts v c) > left (ext v)

```

proof

```

assume assm: ||len v (ts) c|| > 0

```

```

show left ((space ts v) c) < right (ext v) ∧ right ((space ts v) c) > left (ext v)

```

```

proof (rule ccontr)

```

```

  assume ¬(left ((space ts v) c) < right (ext v)
    ∧ right ((space ts v) c) > left (ext v))

```

```

  hence ¬left ((space ts v) c) < right (ext v)
    ∨ ¬right ((space ts v) c) > left (ext v)

```

```

  by best

```

```

  thus False

```

```

proof

```

```

  assume ¬left ((space ts v) c) < right (ext v)

```

```

  hence (left ((space ts v) c) = right (ext v))

```

```

       $\vee$  left ((space ts v) c) > right (ext v)
    by auto
  thus False
proof
  assume left-eq:left ((space ts v) c) = right (ext v)
  hence inside-left:right ((space ts v) c)  $\geq$  left (ext v)
    by (metis order-trans real-int.left-leq-right)
  from left-eq and inside-left have len-v:
    len v ( ts) c = Abs-real-int (max (left (ext v)) (left (space ts v c)),
      min (right (ext v)) (right (space ts v c)))
    using len-def by auto
  hence len v ( ts) c = Abs-real-int (left (space ts v c), left (space ts v c))
    by (metis left-eq max-def min-def real-int.left-leq-right)
  thus False using Abs-real-int-inverse assm real-int.length-def by auto
next
  assume left ((space ts v) c) > right (ext v)
  thus False
    using Abs-real-int-inverse assm len-def real-int.length-def by auto
qed
next
  assume  $\neg$ right ((space ts v) c) > left (ext v)
  hence right ((space ts v) c) = left (ext v)
     $\vee$  right ((space ts v) c) < left (ext v)
    by auto
  thus False
proof
  assume right-eq:right ((space ts v) c) = left (ext v)
  hence inside-right:right (ext v)  $\geq$  left ((space ts v) c)
    by (metis order-trans real-int.left-leq-right)
  from right-eq and inside-right have len-v:
    len v ts c = Abs-real-int (max (left (ext v)) (left (space ts v c)),
      min (right (ext v)) (right (space ts v c)))
    using len-def by auto
  hence
    len v ( ts) c = Abs-real-int(right (space ts v c), right (space ts v c))
    by (metis max commute max-def min-def real-int.left-leq-right right-eq)
  thus False using Abs-real-int-inverse assm real-int.length-def by auto
next
  assume right-le:right ((space ts v) c) < left (ext v)
  thus False
    by (metis (no-types, opaque-lifting) Rep-real-int-inverse assm left.rep-eq
      len-def
        length-zero-iff-borders-eq less-irrefl prod.collapse real-int.rchop-def
        right.rep-eq view.hchop-def view.horizontal-chop-empty-left
        view.horizontal-chop-empty-right)
qed
qed
qed
qed

```

lemma *len-fills-subview*:

$\|len\ v\ ts\ c\| > 0$
 $\longrightarrow (\exists\ v1\ v2\ v3\ v'. (v=v1\|v2) \wedge (v2=v'\|v3) \wedge len\ v'\ ts\ c = ext\ v' \wedge$
 $\|len\ v'\ ts\ c\| = \|len\ v\ ts\ c\|)$

proof

assume *assm*: $\|len\ v\ (ts)\ c\| > 0$

show $\exists\ v1\ v2\ v3\ v'. (v=v1\|v2) \wedge (v2=v'\|v3) \wedge len\ v'\ (ts)\ c = ext\ v' \wedge$
 $\|len\ v'\ (ts)\ c\| = \|len\ v\ (ts)\ c\|$

proof –

from *assm* **have** *inside*:

$left\ ((space\ ts\ v)\ c) < right\ (ext\ v) \wedge right\ ((space\ ts\ v)\ c) > left\ (ext\ v)$

using *len-non-empty-inside* **by** *auto*

hence *len-v*:

$len\ v\ (ts)\ c = Abs-real-int\ (max\ (left\ (ext\ v))\ (left\ (space\ ts\ v\ c)),$
 $min\ (right\ (ext\ v))\ (right\ (space\ ts\ v\ c)))$

using *len-def* **by** *auto*

obtain *v1* **and** *v2* **and** *v3* **and** *v'*

where *v1*:

$v1 = (ext = Abs-real-int(left(ext\ v), left(len\ v\ ts\ c)),$
 $lan = lan\ v,$
 $own = own\ v)$

and *v2*:

$v2 = (ext = Abs-real-int(left(len\ v\ ts\ c), right(ext\ v)),$
 $lan = lan\ v,$
 $own = own\ v)$

and *v'*:

$v' = (ext = Abs-real-int(left(len\ v\ ts\ c), right(len\ v\ ts\ c)),$
 $lan = lan\ v,$
 $own = own\ v)$

and *v3*:

$v3 = (ext = Abs-real-int(right(len\ v\ ts\ c), right(ext\ v)),$
 $lan = lan\ v,$
 $own = own\ v)$

by *blast*

hence *1*: $(v=v1\|v2) \wedge (v2=v'\|v3)$

using *inside* *hchop-def* *real-int.rchop-def* *Abs-real-int-inverse* *real-int.left-leq-right*
 $v1\ v2\ v'\ v3$ *len-def*

by *auto*

have *right*: $right\ (ext\ v') = right\ (len\ v\ ts\ c)$

by (*simp* *add*: *Rep-real-int-inverse* *v'*)

then **have** *right'*: $left\ ((space\ ts\ v)\ c) \leq right\ (ext\ v')$

by (*metis* *inside* *len-space-left* *less-imp-le* *order-trans* *real-int.left-leq-right*)

have *left*: $left\ (ext\ v') = left\ (len\ v\ ts\ c)$

by (*simp* *add*: *Rep-real-int-inverse* *v'*)

then **have** *left'*: $right\ ((space\ ts\ v)\ c) \geq left\ (ext\ v')$

by (*metis* *inside* *len-space-right* *less-imp-le* *order-trans* *real-int.left-leq-right*)

have *inside'*:

$left\ ((space\ ts\ v)\ c) < right\ (ext\ v') \wedge right\ ((space\ ts\ v)\ c) > left\ (ext\ v')$

by (*metis* (*no-types*) *left'* *right'* *antisym-conv* *assm* *inside* *left* *len-space-left* *len-space-right* *less-imp-le* *not-le* *real-int.left-leq-right* *real-int.length-zero-iff-borders-eq* *right*)
have *inside''*:
 $left\ (space\ ts\ v'\ c) < right\ (ext\ v') \wedge right\ (space\ ts\ v'\ c) > left\ (ext\ v')$
using 1 *hchop-def* *inside'* *sensors.space-def* *sensors-axioms*
by *auto*
have *len-v-v'*: $len\ v\ ts\ c = ext\ v'$
by (*metis* *left* *prod.collapse* *right* *left.rep-eq* *right.rep-eq* *Rep-real-int-inverse*)
have $left\ (len\ v\ ts\ c) = max\ (left\ (ext\ v))\ (left\ ((space\ ts\ v)\ c))$
using *len-v* *Abs-real-int-inverse* *Rep-real-int* *inside*
by *auto*
with *left* **have** *left-len'*: $left\ (ext\ v') = max\ (left\ (ext\ v))\ (left\ (space\ ts\ v\ c))$
by *auto*
then **have** *left-len*: $left\ (ext\ v') = max\ (left\ (ext\ v'))\ (left\ (space\ ts\ v'\ c))$
using 1 *hchop-def* *space-def* **by** *fastforce*
have $right\ (len\ v\ ts\ c) = min\ (right\ (ext\ v))\ (right\ ((space\ ts\ v)\ c))$
using *len-v* *Abs-real-int-inverse* *inside* *Rep-real-int* **by** *auto*
with *right* **have** *right-len'*:
 $right\ (ext\ v') = min\ (right\ (ext\ v))\ (right\ (space\ ts\ v\ c))$
by *auto*
then **have** *right-len*:
 $right\ (ext\ v') = min\ (right\ (ext\ v'))\ (right\ (space\ ts\ v'\ c))$
using 1 *hchop-def* *space-def* **by** *fastforce*
have 2: $len\ v'\ (ts)\ c = ext\ v'$
by (*metis* *left-len'* *right-len'* *len-v* *len-v-v'* *order.asym* *inside''* *len-def* *left-len* *right-len*)
have 3: $\|len\ v'\ (ts)\ c\| = \|len\ v\ (ts)\ c\|$
using *len-left* *len-right* *hchop-def*
by (*simp* *add*: 2 *len-v-v'*)
then **show** ?thesis **using** 1 2 3 **by** *blast*
qed
qed

lemma *ext-eq-len-eq*:
 $ext\ v = ext\ v' \wedge own\ v = own\ v' \longrightarrow len\ v\ ts\ c = len\ v'\ ts\ c$
using *left-space* *right-space* *sensors.len-def* *sensors-axioms* **by** *auto*

lemma *len-stable-down*: $(v=v1 \dashv\vdash v2) \longrightarrow len\ v\ ts\ c = len\ v1\ ts\ c$
using *ext-eq-len-eq* *view.vchop-def* **by** *blast*

lemma *len-stable-up*: $(v=v1 \dashv\vdash v2) \longrightarrow len\ v\ ts\ c = len\ v2\ ts\ c$
using *ext-eq-len-eq* *view.vchop-def* **by** *blast*

lemma *len-empty-subview*: $\|len\ v\ ts\ c\| = 0 \wedge (v' \leq v) \longrightarrow \|len\ v'\ ts\ c\| = 0$

proof

assume *assm*: $\|len\ v\ ts\ c\| = 0 \wedge (v' \leq v)$

hence $\exists v1\ v2\ v3\ v4\ v5\ v6. (v=v4\|v1) \wedge (v1=v2\|v3) \wedge (v2=v5 \dashv\vdash v3) \wedge (v3=v' \dashv\vdash v6)$ **using**

```

    somewhere-leq by auto
then obtain v1 v2 v3 vl vr vu vd
  where views:(v=vl||v1) ∧ (v1=v2||vr) ∧ (v2=vd--v3) ∧ (v3=v'--vu)
  by blast
have ||len v1 ts c|| = 0 using views assm len-empty-on-subview2 by blast
hence ||len v2 ts c|| = 0 using views len-empty-on-subview1 by blast
hence ||len v3 ts c|| = 0 using views len-stable-up by auto
thus ||len v' ts c|| = 0 using views len-stable-down by auto
qed

lemma view-leq-len-leq:(ext v ≤ ext v') ∧ (own v = own v') ∧ ||len v ts c|| > 0
  → len v ts c ≤ len v' ts c
  using Abs-real-int-inverse length-def length-ge-zero less-eq-real-int-def
  sensors.len-def sensors.space-def sensors-axioms by auto

end
end

```

10 Basic HMLSL

In this section, we define the basic formulas of HMLSL. All of these basic formulas and theorems are independent of the choice of sensor function. However, they show how the general operators (chop, changes in perspective, atomic formulas) work.

```

theory HMLSL
  imports Restriction Move Length
begin

```

10.1 Syntax of Basic HMLSL

Formulas are functions associating a traffic snapshot and a view with a Boolean value.

```

type-synonym σ = traffic ⇒ view ⇒ bool

```

```

locale hmlsl = restriction+
  fixes sensors::cars ⇒ traffic ⇒ cars ⇒ real
  assumes sensors-ge:(sensors e ts c) > 0 begin
end

```

```

sublocale hmlsl < sensors
  by (simp add: sensors.intro sensors-ge)

```

```

context hmlsl
begin

```

All formulas are defined as abbreviations. As a consequence, proofs will directly refer to the semantics of HMLSL, i.e., traffic snapshots and views.

The first-order operators are direct translations into HOL operators.

abbreviation $mtrue$ $:: \sigma (\top)$
where $\top \equiv \lambda ts w. True$
abbreviation $mfalse$ $:: \sigma (\perp)$
where $\perp \equiv \lambda ts w. False$
abbreviation $mnot$ $:: \sigma \Rightarrow \sigma (\neg [52] 53)$
where $\neg \varphi \equiv \lambda ts w. \neg \varphi(ts)(w)$
abbreviation $mnegpred$ $:: (cars \Rightarrow \sigma) \Rightarrow (cars \Rightarrow \sigma) (\neg [52] 53)$
where $\neg \Phi \equiv \lambda x. \lambda ts w. \neg \Phi(x)(ts)(w)$
abbreviation $mand$ $:: \sigma \Rightarrow \sigma \Rightarrow \sigma$ (**infixr** $\wedge 51$)
where $\varphi \wedge \psi \equiv \lambda ts w. \varphi(ts)(w) \wedge \psi(ts)(w)$
abbreviation mor $:: \sigma \Rightarrow \sigma \Rightarrow \sigma$ (**infix** $\vee 50$)
where $\varphi \vee \psi \equiv \lambda ts w. \varphi(ts)(w) \vee \psi(ts)(w)$
abbreviation $mimp$ $:: \sigma \Rightarrow \sigma \Rightarrow \sigma$ (**infixr** $\rightarrow 49$)
where $\varphi \rightarrow \psi \equiv \lambda ts w. \varphi(ts)(w) \rightarrow \psi(ts)(w)$
abbreviation $mequ$ $:: \sigma \Rightarrow \sigma \Rightarrow \sigma$ (**infixr** $\leftrightarrow 48$)
where $\varphi \leftrightarrow \psi \equiv \lambda ts w. \varphi(ts)(w) \leftrightarrow \psi(ts)(w)$
abbreviation $mforall$ $:: ('a \Rightarrow \sigma) \Rightarrow \sigma (\forall)$
where $\forall \Phi \equiv \lambda ts w. \forall x. \Phi(x)(ts)(w)$
abbreviation $mforallB$ $:: ('a \Rightarrow \sigma) \Rightarrow \sigma$ (**binder** $\forall [8] 9$)
where $\forall x. \varphi(x) \equiv \forall \varphi$
abbreviation $mexists$ $:: ('a \Rightarrow \sigma) \Rightarrow \sigma (\exists)$
where $\exists \Phi \equiv \lambda ts w. \exists x. \Phi(x)(ts)(w)$
abbreviation $mexistsB$ $:: (('a) \Rightarrow \sigma) \Rightarrow \sigma$ (**binder** $\exists [8] 9$)
where $\exists x. \varphi(x) \equiv \exists \varphi$
abbreviation meq $:: 'a \Rightarrow 'a \Rightarrow \sigma$ (**infixr** $= 60$) — Equality
where $x = y \equiv \lambda ts w. x = y$
abbreviation $mgeq$ $:: ('a :: ord) \Rightarrow 'a \Rightarrow \sigma$ (**infix** ≥ 60)
where $x \geq y \equiv \lambda ts w. x \geq y$
abbreviation mgt $:: ('a :: ord) \Rightarrow 'a \Rightarrow \sigma$ (**infix** > 60)
where $x > y \equiv \lambda ts w. x > y$

For the spatial modalities, we use the chopping operations defined on views. Observe that our chop modalities are existential.

abbreviation $hchop$ $:: \sigma \Rightarrow \sigma \Rightarrow \sigma$ (**infixr** $\frown 53$)
where $\varphi \frown \psi \equiv \lambda ts w. \exists v u. (w = v || u) \wedge \varphi(ts)(v) \wedge \psi(ts)(u)$
abbreviation $vchop$ $:: \sigma \Rightarrow \sigma \Rightarrow \sigma$ (**infixr** $\smile 53$)
where $\varphi \smile \psi \equiv \lambda ts w. \exists v u. (w = v -- u) \wedge \varphi(ts)(v) \wedge \psi(ts)(u)$
abbreviation $somewhere$ $:: \sigma \Rightarrow \sigma$ ($\langle - \rangle$ 55)
where $\langle \varphi \rangle \equiv \top \frown (\top \smile \varphi \smile \top) \frown \top$
abbreviation $everywhere$ $:: \sigma \Rightarrow \sigma$ ($[-]$ 55)
where $[\varphi] \equiv \neg \langle \neg \varphi \rangle$

To change the perspective of a view, we use an operator in the fashion of Hybrid Logic.

abbreviation at $:: cars \Rightarrow \sigma \Rightarrow \sigma$ (**@** - - 56)
where $@_c \varphi \equiv \lambda ts w. \forall v'. (w = c > v') \rightarrow \varphi(ts)(v')$

The behavioural modalities are defined as usual modal box-like modalities,

where the accessibility relations are given by the different types of transitions between traffic snapshots.

abbreviation $res\text{-}box::cars \Rightarrow \sigma \Rightarrow \sigma$ ($\Box r'(-)$ - 55)

where $\Box r(c) \varphi \equiv \lambda ts w. \forall ts'. (ts-r(c) \rightarrow ts') \rightarrow \varphi(ts')(w)$

abbreviation $clm\text{-}box::cars \Rightarrow \sigma \Rightarrow \sigma$ ($\Box c'(-)$ - 55)

where $\Box c(c) \varphi \equiv \lambda ts w. \forall ts' n. (ts-c(c,n) \rightarrow ts') \rightarrow \varphi(ts')(w)$

abbreviation $wdr\text{-}box::cars \Rightarrow \sigma \Rightarrow \sigma$ ($\Box wdr'(-)$ - 55)

where $\Box wdr(c) \varphi \equiv \lambda ts w. \forall ts' n. (ts-wdr(c,n) \rightarrow ts') \rightarrow \varphi(ts')(w)$

abbreviation $wdclm\text{-}box::cars \Rightarrow \sigma \Rightarrow \sigma$ ($\Box wdc'(-)$ - 55)

where $\Box wdc(c) \varphi \equiv \lambda ts w. \forall ts'. (ts-wdc(c) \rightarrow ts') \rightarrow \varphi(ts')(w)$

abbreviation $time\text{-}box::\sigma \Rightarrow \sigma$ ($\Box \tau$ - 55)

where $\Box \tau \varphi \equiv \lambda ts w. \forall ts'. (ts \rightsquigarrow ts') \rightarrow \varphi(ts')(move\ ts\ ts'\ w)$

abbreviation $globally::\sigma \Rightarrow \sigma$ (\mathbf{G} - 55)

where $\mathbf{G} \varphi \equiv \lambda ts w. \forall ts'. (ts \Rightarrow ts') \rightarrow \varphi(ts')(move\ ts\ ts'\ w)$

The spatial atoms to refer to reservations, claims and free space are direct translations of the original definitions of MLSL [2] into the Isabelle implementation.

abbreviation $re:: cars \Rightarrow \sigma$ ($re'(-)$ 70)

where

$$re(c) \equiv \lambda ts v. \|ext\ v\| > 0 \wedge len\ v\ ts\ c = ext\ v \wedge \\ restrict\ v\ (res\ ts)\ c = lan\ v \wedge |lan\ v| = 1$$

abbreviation $cl:: cars \Rightarrow \sigma$ ($cl'(-)$ 70)

where

$$cl(c) \equiv \lambda ts v. \|ext\ v\| > 0 \wedge len\ v\ ts\ c = ext\ v \wedge \\ restrict\ v\ (clm\ ts)\ c = lan\ v \wedge |lan\ v| = 1$$

abbreviation $free:: \sigma$ ($free$)

where

$$free \equiv \lambda ts v. \|ext\ v\| > 0 \wedge |lan\ v| = 1 \wedge \\ (\forall c. \|len\ v\ ts\ c\| = 0 \vee \\ (restrict\ v\ (clm\ ts)\ c = \emptyset \wedge restrict\ v\ (res\ ts)\ c = \emptyset))$$

Even though we do not need them for the subsequent proofs of safety, we define ways to measure the number of lanes (width) and the size of the extension (length) of a view. This allows us to connect the atomic formulas for reservations and claims with the atom denoting free space [5].

abbreviation $width\text{-}eq::nat \Rightarrow \sigma$ ($\omega =$ - 60)

where $\omega = n \equiv \lambda ts v. |lan\ v| = n$

abbreviation $width\text{-}geq::nat \Rightarrow \sigma$ ($\omega \geq$ - 60)

where $\omega \geq n \equiv \lambda ts v. |lan\ v| \geq n$

abbreviation $width\text{-}gt::nat \Rightarrow \sigma$ ($\omega >$ - 60)

where $\omega > n \equiv (\omega = n+1) \smile \top$

abbreviation $length\text{-}eq::real \Rightarrow \sigma$ ($\mathbf{1} =$ - 60)

where $\mathbf{l} = r \equiv \lambda ts v. \|ext v\| = r$

abbreviation *length-ge*:: $real \Rightarrow \sigma$ ($\mathbf{l} > - 60$)
where $\mathbf{l} > r \equiv \lambda ts v. \|ext v\| > r$

abbreviation *length-geq*:: $real \Rightarrow \sigma$ ($\mathbf{l} \geq - 60$)
where $\mathbf{l} \geq r \equiv (\mathbf{l} = r) \vee (\mathbf{l} > r)$

For convenience, we use abbreviations for the validity and satisfiability of formulas. While the former gives a nice way to express theorems, the latter is useful within proofs.

abbreviation *valid* :: $\sigma \Rightarrow bool$ ($\models - 10$)
where $\models \varphi \equiv \forall ts. \forall v. \varphi(ts)(v)$

abbreviation *satisfies*:: $traffic \Rightarrow view \Rightarrow \sigma \Rightarrow bool$ ($- , - \models - 10$)
where $ts, v \models \varphi \equiv \varphi(ts)(v)$

10.2 Theorems about Basic HMLSL

lemma *hchop-weaken1*: $\models \varphi \rightarrow (\varphi \frown \top)$
using *horizontal-chop-empty-right* **by** *fastforce*

lemma *hchop-weaken2*: $\models \varphi \rightarrow (\top \frown \varphi)$
using *horizontal-chop-empty-left* **by** *fastforce*

lemma *hchop-weaken*: $\models \varphi \rightarrow (\top \frown \varphi \frown \top)$
using *hchop-weaken1 hchop-weaken2* **by** *metis*

lemma *hchop-neg1*: $\models \neg (\varphi \frown \top) \rightarrow ((\neg \varphi) \frown \top)$
using *horizontal-chop1* **by** *fastforce*

lemma *hchop-neg2*: $\models \neg (\top \frown \varphi) \rightarrow (\top \frown \neg \varphi)$
using *horizontal-chop1* **by** *fastforce*

lemma *hchop-disj-distr1*: $\models ((\varphi \frown (\psi \vee \chi)) \leftrightarrow ((\varphi \frown \psi) \vee (\varphi \frown \chi)))$
by *blast*

lemma *hchop-disj-distr2*: $\models (((\psi \vee \chi) \frown \varphi) \leftrightarrow ((\psi \frown \varphi) \vee (\chi \frown \varphi)))$
by *blast*

lemma *hchop-assoc*: $\models \varphi \frown (\psi \frown \chi) \leftrightarrow (\varphi \frown \psi) \frown \chi$
using *horizontal-chop-assoc1 horizontal-chop-assoc2* **by** *fastforce*

lemma *v-chop-weaken1*: $\models (\varphi \rightarrow (\varphi \smile \top))$
using *vertical-chop-empty-down* **by** *fastforce*

lemma *v-chop-weaken2*: $\models (\varphi \rightarrow (\top \smile \varphi))$
using *vertical-chop-empty-up* **by** *fastforce*

lemma *v-chop-assoc*: $\models (\varphi \smile (\psi \smile \chi)) \leftrightarrow ((\varphi \smile \psi) \smile \chi)$
using *vertical-chop-assoc1 vertical-chop-assoc2* **by** *fastforce*

lemma *vhchop-disj-distr1*: $\models ((\varphi \smile (\psi \vee \chi)) \leftrightarrow ((\varphi \smile \psi) \vee (\varphi \smile \chi)))$
by *blast*

lemma *vhchop-disj-distr2*: $\models (((\psi \vee \chi) \smile \varphi) \leftrightarrow ((\psi \smile \varphi) \vee (\chi \smile \varphi)))$
by *blast*

lemma *at-exists* : $\models \varphi \rightarrow (\exists c. @c \varphi)$
proof (*rule allI|rule impI*)
fix *ts v*
assume *assm:ts,v* $\models \varphi$
obtain *d* **where** *d-def:d=own v* **by** *blast*
then have *ts,v* $\models @d \varphi$ **using** *assm switch-refl switch-unique* **by** *fastforce*
thus *ts,v* $\models (\exists c. @c \varphi)$..
qed

lemma *at-conj-distr*: $\models (@c (\varphi \wedge \psi)) \leftrightarrow ((@c \varphi) \wedge (@c \psi))$
using *switch-unique* **by** *blast*

lemma *at-disj-dist*: $\models (@c (\varphi \vee \psi)) \leftrightarrow ((@c \varphi) \vee (@c \psi))$
using *switch-unique* **by** *fastforce*

lemma *at-hchop-dist1*: $\models (@c (\varphi \frown \psi)) \rightarrow ((@c \varphi) \frown (@c \psi))$
proof (*rule allI|rule impI*)
fix *ts v*
assume *assm:ts, v* $\models (@c (\varphi \frown \psi))$
obtain *v'* **where** *v':v=c>v'* **using** *switch-always-exists* **by** *fastforce*
with *assm* **obtain** *v1'* **and** *v2'*
where *chop:(v'=v1' || v2')* $\wedge (ts, v1' \models \varphi) \wedge (ts, v2' \models \psi)$
by *blast*
from *chop* **and** *v'* **obtain** *v1* **and** *v2*
where *origin:(v1=c>v1')* $\wedge (v2=c>v2') \wedge (v=v1 || v2)$
using *switch-hchop2* **by** *fastforce*
hence *v1:ts,v1* $\models (@c \varphi)$ **and** *v2:ts,v2* $\models (@c \psi)$
using *switch-unique chop* **by** *fastforce+*
from *v1* **and** *v2* **and** *origin* **show** *ts,v* $\models (@c \varphi) \frown (@c \psi)$ **by** *blast*
qed

lemma *at-hchop-dist2*: $\models ((@c \varphi) \frown (@c \psi)) \rightarrow (@c (\varphi \frown \psi))$
using *switch-unique switch-hchop1 switch-def* **by** *metis*

lemma *at-hchop-dist*: $\models ((@c \varphi) \frown (@c \psi)) \leftrightarrow (@c (\varphi \frown \psi))$
using *at-hchop-dist1 at-hchop-dist2* **by** *blast*

lemma *at-vchop-dist1*: $\models (@c (\varphi \smile \psi)) \rightarrow ((@c \varphi) \smile (@c \psi))$
proof (*rule allI|rule impI*)
fix *ts v*

assume $assm:ts, v \models (@c (\varphi \smile \psi))$
obtain v' **where** $v':v=c>v'$ **using** *switch-always-exists* **by** *fastforce*
with *assm* **obtain** $v1'$ **and** $v2'$
where $chop:(v'=v1'--v2') \wedge (ts,v1' \models \varphi) \wedge (ts,v2' \models \psi)$
by *blast*
from *chop* **and** v' **obtain** $v1$ **and** $v2$
where $origin:(v1=c>v1') \wedge (v2=c>v2') \wedge (v=v1--v2)$
using *switch-vchop2* **by** *fastforce*
hence $v1:ts,v1 \models (@c \varphi)$ **and** $v2:ts,v2 \models (@c \psi)$
using *switch-unique chop* **by** *fastforce+*
from $v1$ **and** $v2$ **and** *origin* **show** $ts,v \models (@c \varphi) \smile (@c \psi)$ **by** *blast*
qed

lemma *at-vchop-dist2*: $\models (@c \varphi) \smile (@c \psi) \rightarrow (@c (\varphi \smile \psi))$
using *switch-unique switch-vchop1 switch-def* **by** *metis*

lemma *at-vchop-dist*: $\models (@c \varphi) \smile (@c \psi) \leftrightarrow (@c (\varphi \smile \psi))$
using *at-vchop-dist1 at-vchop-dist2* **by** *blast*

lemma *at-eq*: $\models (@e c = d) \leftrightarrow (c = d)$
using *switch-always-exists* **by** (*metis*)

lemma *at-neg1*: $\models (@c \neg \varphi) \rightarrow \neg (@c \varphi)$
using *switch-unique*
by (*metis select-convs switch-def*)

lemma *at-neg2*: $\models \neg (@c \varphi) \rightarrow (@c \neg \varphi)$
using *switch-unique* **by** *fastforce*

lemma *at-neg*: $\models (@c(\neg \varphi)) \leftrightarrow \neg (@c \varphi)$
using *at-neg1 at-neg2* **by** *metis*

lemma *at-neg'*: $ts,v \models \neg (@c \varphi) \leftrightarrow (@c(\neg \varphi))$ **using** *at-neg* **by** *simp*

lemma *at-neg-neg1*: $\models (@c \varphi) \rightarrow \neg(@c \neg \varphi)$
using *switch-unique switch-def switch-refl*
by (*metis select-convs switch-def*)

lemma *at-neg-neg2*: $\models \neg(@c \neg \varphi) \rightarrow (@c \varphi)$
using *switch-unique switch-def switch-refl*
by *metis*

lemma *at-neg-neg*: $\models (@c \varphi) \leftrightarrow \neg(@c \neg \varphi)$
using *at-neg-neg1 at-neg-neg2* **by** *metis*

lemma *globally-all-iff*: $\models (\mathbf{G}(\forall c. \varphi)) \leftrightarrow (\forall c. (\mathbf{G} \varphi))$ **by** *simp*

lemma *globally-all-iff'*: $ts,v \models (\mathbf{G}(\forall c. \varphi)) \leftrightarrow (\forall c. (\mathbf{G} \varphi))$ **by** *simp*

lemma *globally-refl*: $\models (\mathbf{G} \varphi) \rightarrow \varphi$

using *traffic.abstract.refl traffic.move-nothing* by *fastforce*

lemma *globally-4*: $\models (\mathbf{G} \ \varphi) \rightarrow \mathbf{G} \ \mathbf{G} \ \varphi$
proof (*rule allI|rule impI*)
 fix $ts \ v \ ts' \ ts''$
 assume $1:ts \Rightarrow ts'$ and $2:ts' \Rightarrow ts''$ and $3:ts,v \models \mathbf{G} \ \varphi$
 from 2 and 1 have $ts \Rightarrow ts''$ using *traffic.abs-trans* by *blast*
 moreover from 1 and 2 have $move \ ts' \ ts'' (move \ ts \ ts' \ v) = move \ ts \ ts'' \ v$
 using *traffic.move-trans* by *blast*
 with 3 show $ts'', move \ ts' \ ts'' (move \ ts \ ts' \ v) \models \varphi$ using *calculation* by *simp*
qed

lemma *spatial-weaken*: $\models (\varphi \rightarrow \langle \varphi \rangle)$
 using *horizontal-chop-empty-left horizontal-chop-empty-right vertical-chop-empty-down*
vertical-chop-empty-up
 by *fastforce*

lemma *spatial-weaken2*: $\models (\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \langle \psi \rangle)$
 using *spatial-weaken horizontal-chop-empty-left horizontal-chop-empty-right*
vertical-chop-empty-down vertical-chop-empty-up
 by *blast*

lemma *somewhere-distr*: $\models \langle \varphi \vee \psi \rangle \leftrightarrow \langle \varphi \rangle \vee \langle \psi \rangle$
 by *blast*

lemma *somewhere-and*: $\models \langle \varphi \wedge \psi \rangle \rightarrow \langle \varphi \rangle \wedge \langle \psi \rangle$
 by *blast*

lemma *somewhere-and-or-distr* : $\models (\langle \chi \wedge (\varphi \vee \psi) \rangle \leftrightarrow \langle \chi \wedge \varphi \rangle \vee \langle \chi \wedge \psi \rangle)$
 by *blast*

lemma *width-add1*: $\models ((\omega = x) \smile (\omega = y) \rightarrow \omega = x+y)$
 using *vertical-chop-add1* by *fastforce*

lemma *width-add2*: $\models ((\omega = x+y) \rightarrow (\omega = x) \smile \omega = y)$
 using *vertical-chop-add2* by *fastforce*

lemma *width-hchop-stable*: $\models ((\omega = x) \leftrightarrow ((\omega = x) \frown (\omega = x)))$
 using *hchop-def horizontal-chop1*
 by *force*

lemma *length-geq-zero*: $\models (\mathbf{1} \geq 0)$
 by (*metis order.not-eq-order-implies-strict real-int.length-ge-zero*)

lemma *length-split*: $\models ((\mathbf{1} > 0) \rightarrow (\mathbf{1} > 0) \frown (\mathbf{1} > 0))$
 using *horizontal-chop-non-empty* by *fastforce*

lemma *length-meld*: $\models ((\mathbf{1} > 0) \frown (\mathbf{1} > 0) \rightarrow (\mathbf{1} > 0))$

```

using hchop-def real-int.chop-add-length-ge-0
by (metis (no-types, lifting))

lemma length-dense: $\models((\mathbf{1} > 0) \leftrightarrow (\mathbf{1} > 0) \frown (\mathbf{1} > 0))$ 
using length-meld length-split by blast

lemma length-add1: $\models((\mathbf{1}=x) \frown (\mathbf{1}=y)) \rightarrow (\mathbf{1}=x+y)$ 
using hchop-def real-int.rchop-def real-int.length-def by fastforce

lemma length-add2: $\models(x \geq 0 \wedge y \geq 0) \rightarrow ((\mathbf{1}=x+y) \rightarrow ((\mathbf{1}=x) \frown (\mathbf{1}=y)))$ 
using horizontal-chop-split-add by fastforce

lemma length-add: $\models(x \geq 0 \wedge y \geq 0) \rightarrow ((\mathbf{1}=x+y) \leftrightarrow ((\mathbf{1}=x) \frown (\mathbf{1}=y)))$ 
using length-add1 length-add2 by blast

lemma length-vchop-stable: $\models(\mathbf{1} = x) \leftrightarrow ((\mathbf{1} = x) \smile (\mathbf{1} = x))$ 
using vchop-def vertical-chop1 by fastforce

lemma res-ge-zero: $\models(\text{re}(c) \rightarrow \mathbf{1} > 0)$ 
by blast

lemma clm-ge-zero: $\models(\text{cl}(c) \rightarrow \mathbf{1} > 0)$ 
by blast

lemma free-ge-zero: $\models(\text{free} \rightarrow \mathbf{1} > 0)$ 
by blast

lemma width-res: $\models(\text{re}(c) \rightarrow \omega = 1)$ 
by auto

lemma width-clm: $\models(\text{cl}(c) \rightarrow \omega = 1)$ 
by simp

lemma width-free: $\models(\text{free} \rightarrow \omega = 1)$ 
by simp

lemma width-somewhere-res: $\models \langle \text{re}(c) \rangle \rightarrow (\omega \geq 1)$ 
proof (rule allI|rule impI)+
  fix ts v
  assume ts,v  $\models \langle \text{re}(c) \rangle$ 
  then show ts,v  $\models (\omega \geq 1)$ 
    using view.hchop-def view.vertical-chop-width-mon by fastforce
qed

lemma clm-disj-res: $\models \neg \langle \text{cl}(c) \wedge \text{re}(c) \rangle$ 
proof (rule allI|rule notI)+
  fix ts v
  assume ts,v  $\models \langle \text{cl}(c) \wedge \text{re}(c) \rangle$ 
  then obtain v' where  $v' \leq v \wedge (\text{ts},v' \models \text{cl}(c) \wedge \text{re}(c))$ 

```

by (*meson view.somewhere-leq*)
then show *False using disjoint*
by (*metis card-non-empty-geq-one inf.idem restriction.restriction-clm-leq-one*
restriction.restriction-clm-res-disjoint)
qed

lemma *width-ge*: $\models (\omega > 0) \rightarrow (\exists x. (\omega = x) \wedge (x > 0))$
using *vertical-chop-add1 add-gr-0 zero-less-one* **by** *auto*

lemma *two-res-width*: $\models ((re(c) \smile re(c)) \rightarrow \omega = 2)$
by (*metis one-add-one width-add1*)

lemma *res-at-most-two*: $\models \neg (re(c) \smile re(c) \smile re(c))$
proof (*rule allI | rule notI*) +
fix *ts v*
assume *ts, v* $\models (re(c) \smile re(c) \smile re(c))$
then obtain *v'* **and** *v1* **and** *v2* **and** *v3*
where *v = v1 -- v'* **and** *v' = v2 -- v3*
and *ts, v1* $\models re(c)$ **and** *ts, v2* $\models re(c)$ **and** *ts, v3* $\models re(c)$
by *blast*
then show *False*
proof –
have $|restrict\ v'\ (res\ ts)\ c| < |restrict\ v\ (res\ ts)\ c|$
using $\langle ts, v1 \models re(c) \rangle \langle v = v1 -- v' \rangle$ *restriction.restriction-add-res* **by** *auto*
then show *?thesis*
by (*metis (no-types) \langle ts, v2 \models re(c) \rangle \langle ts, v3 \models re(c) \rangle \langle v' = v2 -- v3 \rangle not-less*
one-add-one restriction-add-res restriction-res-leq-two)

qed
qed

lemma *res-at-most-two2*: $\models \neg \langle re(c) \smile re(c) \smile re(c) \rangle$
using *res-at-most-two* **by** *blast*

lemma *res-at-most-somewhere*: $\models \neg \langle re(c) \rangle \smile \langle re(c) \rangle \smile \langle re(c) \rangle$
proof (*rule allI | rule notI*) +
fix *ts v*
assume *asm: ts, v* $\models (\langle re(c) \rangle \smile \langle re(c) \rangle \smile \langle re(c) \rangle)$
obtain *vu* **and** *v1* **and** *vm* **and** *vd*
where *chops: (v = vu -- v1) \wedge (v1 = vm -- vd) \wedge (ts, vu $\models \langle re(c) \rangle$)*
 $\wedge (ts, vm \models \langle re(c) \rangle) \wedge (ts, vd \models \langle re(c) \rangle)$
using *asm* **by** *blast*
from *chops* **have** *res-vu: |restrict vu (res ts) c| \geq 1*
by (*metis restriction-card-somewhere-mon*)
from *chops* **have** *res-vm: |restrict vm (res ts) c| \geq 1*
by (*metis restriction-card-somewhere-mon*)
from *chops* **have** *res-vd: |restrict vd (res ts) c| \geq 1*
by (*metis restriction-card-somewhere-mon*)
from *chops* **have**

```

|restrict v (res ts) c| =
|restrict vu (res ts) c| + |restrict vm (res ts) c| + |restrict vd (res ts) c|
using restriction-add-res by force
with res-vu and res-vd res-vm have |restrict v (res ts) c| ≥ 3
by linarith
with restriction-res-leq-two show False
by (metis not-less-eq-eq numeral-2-eq-2 numeral-3-eq-3)
qed

lemma res-adj:|=¬ (re(c) ∪ (ω > 0) ∪ re(c))
proof (rule allI|rule notI)+
fix ts v
assume ts,v |= (re(c) ∪ (ω > 0) ∪ re(c))
then obtain v1 and v' and v2 and vn
where chop:(v=v1--v') ∧ (v'=vn--v2) ∧ (ts,v1|=re(c))
      ∧ (ts,vn |= ω > 0) ∧ (ts,v2|=re(c))
by blast
hence res1:|restrict v1 (res ts) c| ≥ 1 by (simp add: le-numeral-extra(4))
from chop have res2:|restrict v2 (res ts) c| ≥ 1 by (simp add: le-numeral-extra(4))
from res1 and res2 have |restrict v (res ts) c| ≥ 2
using chop restriction.restriction-add-res by auto
then have resv:|restrict v (res ts) c| = 2
using dual-order.antisym restriction.restriction-res-leq-two by blast
hence res-two-lanes:|res ts c| = 2 using atMostTwoRes restrict-res
by (metis (no-types, lifting) nat-int.card-subset-le dual-order.antisym)
from this obtain p where p-def:Rep-nat-int (res ts c) = {p, p+1}
using consecutiveRes by force
have Abs-nat-int {p,p+1} ⊆ lan v
by (metis Rep-nat-int-inverse atMostTwoRes card-seteq finite-atLeastAtMost
insert-not-empty nat-int.card'.rep-eq nat-int.card-seq less-eq-nat-int.rep-eq
p-def resv restrict-res restrict-view)
have vn-not-e:lan vn ≠ ∅ using chop
by (metis nat-int.card-empty-zero less-irrefl width-ge)
hence consec-vn-v2:nat-int.consec (lan vn) (lan v2)
using nat-int.card-empty-zero chop nat-int.nchop-def one-neq-zero vchop-def
by auto
have v'-not-e:lan v' ≠ ∅ using chop
by (metis less-irrefl nat-int.card-empty-zero view.vertical-chop-assoc2 width-ge)
hence consec-v1-v':nat-int.consec (lan v1) (lan v')
by (metis (no-types, lifting) nat-int.card-empty-zero chop nat-int.nchop-def
one-neq-zero vchop-def)
hence consec-v1-vn:nat-int.consec (lan v1) (lan vn)
by (metis (no-types, lifting) chop consec-vn-v2 nat-int.consec-def
nat-int.chop-min vchop-def)
hence lesser-con:∀ n m. (n ∈ (lan v1) ∧ m ∈ (lan v2) → n < m)
using consec-v1-vn consec-vn-v2 nat-int.consec-trans-lesser
by auto
have p-in-v1:p ∈ lan v1
proof (rule ccontr)

```

assume $\neg p \in \text{lan } v1$
then have $p \notin \text{lan } v1$ **by** (*simp*)
hence $p \notin \text{restrict } v1 \text{ (res } ts) \ c$ **using** *chop* **by** (*simp add: chop*)
then have $p+1 \in \text{restrict } v1 \text{ (res } ts) \ c$
proof –
have $\{p, p+1\} \cap (\text{Rep-nat-int (res } ts) \ c) \cap \text{Rep-nat-int (lan } v1) \neq \{\}$
by (*metis chop Rep-nat-int-inject bot-nat-int.rep-eq consec-v1-v'*
inf-nat-int.rep-eq nat-int.consec-def p-def restriction.restrict-def)
then have $p+1 \in \text{Rep-nat-int (lan } v1)$
using $\langle p \notin \text{restrict } v1 \text{ (res } ts) \ c \rangle$ *inf-nat-int.rep-eq not-in.rep-eq*
restriction.restrict-def **by** *force*
then show *?thesis*
using *chop el.rep-eq* **by** *presburger*
qed
hence $\text{succ } p \in \text{lan } v1$ **using** *chop* **by** (*simp add: chop*)
hence $p+1 \notin \text{lan } v2$ **using** *p-def restrict-def* **using** *lesser-con nat-int.el.rep-eq*
nat-int.not-in.rep-eq **by** *auto*
then have $p \in \text{restrict } v2 \text{ (res } ts) \ c$
proof –
have $f1: \text{minimum (lan } v2) \in \text{Rep-nat-int (lan } v2)$
using *consec-vn-v2 el.rep-eq minimum-in nat-int.consec-def* **by** *simp*
have $\text{lan } v2 \sqsubseteq \text{res } ts \ c$
by (*metis (no-types) chop restriction.restrict-res*)
then have $\text{minimum (lan } v2) = p$
using $\langle p+1 \notin \text{lan } v2 \rangle$ *f1 less-eq-nat-int.rep-eq not-in.rep-eq p-def* **by** *auto*
then show *?thesis*
using *f1* **by** (*metis chop el.rep-eq*)
qed
hence $p \in \text{lan } v2$ **using** *p-def restrict-def*
using *chop* **by** *auto*
show *False* **using** *lesser-con succ-p p* **by** *blast*
qed
hence $p \notin \text{lan } v2$ **using** *p-def restrict-def* *lesser-con*
nat-int.el.rep-eq nat-int.not-in.rep-eq
by *auto*
then have $p+1 \in \text{restrict } v2 \text{ (res } ts) \ c$
proof –
have $f1: \text{minimum (lan } v2) \in \text{lan } v2$
using *consec-vn-v2 minimum-in nat-int.consec-def* **by** *simp*
obtain x **where** $\text{mini } x = \text{minimum (lan } v2)$ **by** *blast*
have $x = p+1$
by (*metis IntD1 p-not-in-v2 chop el.rep-eq f1 inf-nat-int.rep-eq insertE mini*
not-in.rep-eq p-def restriction.restrict-def singletonD)
then show *?thesis*
using *chop f1 mini* **by** *auto*
qed
hence $\text{succ } p \in \text{lan } v2$ **using** *p-def restrict-def* **using** *chop* **by** *auto*
have $\forall n \ m. (n \in (\text{lan } v1) \wedge m \in (\text{lan } vn) \longrightarrow n < m)$
using *consec-v1-vn nat-int.consec-lesser* **by** *auto*

with $p\text{-in-}v1$ **have** $ge\text{-}p:\forall m. (m \in \text{lan } vn \longrightarrow p < m)$
by *blast*
have $\forall n m. (n \in (\text{lan } vn) \wedge m \in (\text{lan } v2) \longrightarrow n < m)$
using *consec-vn-v2 nat-int.consec-lesser* **by** *auto*
with $suc\text{-}p\text{-in-}v2$ **have** $less\text{-}suc\text{-}p:\forall m. (m \in \text{lan } vn \longrightarrow m < p+1)$
by *blast*
have $\forall m. (m \in \text{lan } vn \longrightarrow (m < p+1 \wedge m > p))$
using $ge\text{-}p$ $less\text{-}suc\text{-}p$ **by** *auto*
hence $\neg(\exists m. (m \in \text{lan } vn))$
by (*metis One-nat-def Suc-leI add.right-neutral add-Suc-right linorder-not-less*)
hence $\text{lan } vn = \emptyset$ **using** *nat-int.non-empty-elem-in* **by** *auto*
with $vn\text{-not-}e$ **show** *False* **by** *blast*
qed

lemma $clm\text{-}sing:\models\neg (cl(c) \smile cl(c))$
using *atMostOneClm restriction-add-clm vchop-def restriction-clm-leq-one*
by (*metis (no-types, opaque-lifting) add-eq-self-zero le-add1 le-antisym one-neq-zero*)

lemma $clm\text{-}sing\text{-}somewhere:\models\neg \langle cl(c) \smile cl(c) \rangle$
using *clm-sing* **by** *blast*

lemma $clm\text{-}sing\text{-}not\text{-}interrupted:\models\neg (cl(c) \smile \top \smile cl(c))$
using *atMostOneClm restriction-add-clm vchop-def restriction-clm-leq-one clm-sing*
by (*metis (no-types, opaque-lifting) add.commute add-eq-self-zero dual-order.antisym le-add1 one-neq-zero*)

lemma $clm\text{-}sing\text{-}somewhere2:\models\neg (\top \smile cl(c) \smile \top \smile cl(c) \smile \top)$
using *clm-sing-not-interrupted vertical-chop-assoc1*
by *meson*

lemma $clm\text{-}sing\text{-}somewhere3:\models\neg \langle (\top \smile cl(c) \smile \top \smile cl(c) \smile \top) \rangle$
by (*meson clm-sing-not-interrupted view.vertical-chop-assoc1*)

lemma $clm\text{-}at\text{-}most\text{-}somewhere:\models\neg (\langle cl(c) \rangle \smile \langle cl(c) \rangle)$

proof (*rule allI | rule notI*)+

fix $ts\ v$

assume $asm:ts,v \models (\langle cl(c) \rangle \smile \langle cl(c) \rangle)$

obtain vu **and** vd

where $chops:(v=vu--vd) \wedge (ts,vu \models \langle cl(c) \rangle) \wedge (ts,vd \models \langle cl(c) \rangle)$

using asm **by** *blast*

from $chops$ **have** $clm\text{-}vu:|restrict\ vu\ (clm\ ts)\ c| \geq 1$

by (*metis restriction-card-somewhere-mon*)

from $chops$ **have** $clm\text{-}vd:|restrict\ vd\ (clm\ ts)\ c| \geq 1$

by (*metis restriction-card-somewhere-mon*)

from $chops$ **have** $clm\text{-}add:$

$|restrict\ v\ (clm\ ts)\ c| = |restrict\ vu\ (clm\ ts)\ c| + |restrict\ vd\ (clm\ ts)\ c|$

using *restriction-add-clm* **by** *auto*

with $clm\text{-}vu$ **and** $clm\text{-}vd$ **have** $|restrict\ v\ (clm\ ts)\ c| \geq 2$

using *add.commute add-eq-self-zero dual-order.antisym le-add1 less-one not-le*

restriction-res-leq-two
 by *linarith*
 with *restriction-clm-leq-one* show *False*
 by (*metis One-nat-def not-less-eq-eq numeral-2-eq-2*)
 qed

lemma *res-decompose*: $\models (re(c) \rightarrow re(c) \frown re(c))$
proof (*rule allI* | *rule impI*) +
 fix *ts v*
 assume *asm:ts,v* $\models re(c)$
 then obtain *v1* and *v2*
 where $1:v=v1 \parallel v2$ and $2:\|ext\ v1\| > 0$ and $3:\|ext\ v2\| > 0$
 using *view.horizontal-chop-non-empty* by *blast*
 then have $4:|lan\ v1| = 1$ and $5:|lan\ v2| = 1$
 using *asm view.hchop-def* by *auto*
 then have $6:ts,v1 \models re(c)$
 by (*metis 2 1 asm len-view-hchop-left restriction.restrict-eq-lan-subst*
restriction.restrict-view restriction.restriction-stable1)
 from *5* have $7:ts,v2 \models re(c)$
 by (*metis 1 3 6 asm len-view-hchop-right restriction.restrict-eq-lan-subst*
restriction.restrict-view restriction.restriction-stable)
 from *1* and *6* and *7* show $ts,v \models re(c) \frown re(c)$ by *blast*
 qed

lemma *res-compose*: $\models (re(c) \frown re(c) \rightarrow re(c))$
 using *real-int.chop-dense len-compose-hchop hchop-def length-dense restrict-def*
 by (*metis (no-types, lifting)*)

lemma *res-dense*: $\models re(c) \leftrightarrow re(c) \frown re(c)$
 using *res-decompose res-compose* by *blast*

lemma *res-continuous* : $\models (re(c) \rightarrow (\neg (\top \frown (\neg re(c) \wedge 1 > 0) \frown \top)))$
 by (*metis (no-types, lifting) hchop-def len-view-hchop-left len-view-hchop-right*
restrict-def)

lemma *no-clm-before-res*: $\models \neg (cl(c) \frown re(c))$
 by (*metis (no-types, lifting) nat-int.card-empty-zero nat-int.card-subset-le*
disjoint hchop-def inf-assoc inf-le1 not-one-le-zero restrict-def)

lemma *no-clm-before-res2*: $\models \neg (cl(c) \frown \top \frown re(c))$
proof (*rule ccontr*)
 assume $\neg (\models \neg (cl(c) \frown \top \frown re(c)))$
 then obtain *ts* and *v* where *asm:ts,v* $\models (cl(c) \frown \top \frown re(c))$ by *blast*
 then have *clm-subst:restrict v (clm ts) c = restrict v (res ts) c*
 using *restriction-stable*
 by (*metis (no-types, lifting) hchop-def restrict-def*)

have $\text{restrict } v \text{ (clm } ts) c \neq \emptyset$
using *assm nat-int.card-non-empty-geq-one restriction-stable1*
by *auto*
then have $\text{res-in-neq:restrict } v \text{ (clm } ts) c \sqcap \text{restrict } v \text{ (res } ts) c \neq \emptyset$
using *clm-subs inf-absorb1*
by (*simp*)
then show *False* **using** *restriction-clm-res-disjoint*
by (*metis inf-commute restriction.restriction-clm-res-disjoint*)
qed

lemma *clm-decompose*: $\models (\text{cl}(c) \rightarrow \text{cl}(c) \frown \text{cl}(c))$
proof (*rule allI | rule impI*) +
fix *ts v*
assume *assm*: $ts, v \models \text{cl}(c)$
have *restr*: $\text{restrict } v \text{ (clm } ts) c = \text{lan } v$ **using** *assm* **by** *simp*
have *len-ge-zero*: $\|\text{len } v \text{ ts } c\| > 0$ **using** *assm* **by** *simp*
have *len*: $\text{len } v \text{ ts } c = \text{ext } v$ **using** *assm* **by** *simp*
obtain *v1 v2* **where** *chop*: $(v=v1 \parallel v2) \wedge \|\text{ext } v1\| > 0 \wedge \|\text{ext } v2\| > 0$
using *assm view.horizontal-chop-non-empty*
using *length-split* **by** *blast*
from *chop* **and** *len* **have** *len-v1*: $\text{len } v1 \text{ ts } c = \text{ext } v1$
using *len-view-hchop-left* **by** *blast*
from *chop* **and** *len* **have** *len-v2*: $\text{len } v2 \text{ ts } c = \text{ext } v2$
using *len-view-hchop-right* **by** *blast*
from *chop* **and** *restr* **have** *restr-v1*: $\text{restrict } v1 \text{ (clm } ts) c = \text{lan } v1$
by (*metis (no-types, lifting) hchop-def restriction.restriction-stable1*)
from *chop* **and** *restr* **have** *restr-v2*: $\text{restrict } v2 \text{ (clm } ts) c = \text{lan } v2$
by (*metis (no-types, lifting) hchop-def restriction.restriction-stable2*)
from *chop* **and** *len-v1 len-v2 restr-v1 restr-v2* **show** $ts, v \models \text{cl}(c) \frown \text{cl}(c)$
using *hchop-def assm* **by** *force*
qed

lemma *clm-compose*: $\models (\text{cl}(c) \frown \text{cl}(c) \rightarrow \text{cl}(c))$
using *real-int.chop-dense len-compose-hchop hchop-def length-dense restrict-def*
by (*metis (no-types, lifting)*)

lemma *clm-dense*: $\models \text{cl}(c) \leftrightarrow \text{cl}(c) \frown \text{cl}(c)$
using *clm-decompose clm-compose* **by** *blast*

lemma *clm-continuous* : $\models (\text{cl}(c) \rightarrow (\neg (\top \frown (\neg \text{cl}(c) \wedge \mathbf{1} > 0) \frown \top)))$
by (*metis (no-types, lifting) hchop-def len-view-hchop-left len-view-hchop-right restrict-def*)

lemma *res-not-free*: $\models (\exists c. \text{re}(c) \rightarrow \neg \text{free})$
using *nat-int.card-empty-zero one-neq-zero* **by** *auto*

lemma *clm-not-free*: $\models (\exists c. cl(c) \rightarrow \neg free)$
using *nat-int.card-empty-zero* **by** *auto*

lemma *free-no-res*: $\models (free \rightarrow \neg(\exists c. re(c)))$
using *nat-int.card-empty-zero one-neq-zero*
by (*metis less-irrefl*)

lemma *free-no-clm*: $\models (free \rightarrow \neg(\exists c. cl(c)))$
using *nat-int.card-empty-zero one-neq-zero* **by** (*metis less-irrefl*)

lemma *free-decompose*: $\models free \rightarrow (free \frown free)$
proof (*rule allI|rule impI*)
fix *ts v*
assume *asm:ts,v* $\models free$
obtain *v1 and v2*
where *non-empty-v1-v2*: $(v=v1 \parallel v2) \wedge \|ext\ v1\| > 0 \wedge \|ext\ v2\| > 0$
using *asm length-dense* **by** *blast*
have *one-lane*: $|lan\ v1| = 1 \wedge |lan\ v2| = 1$
using *asm hchop-def non-empty-v1-v2*
by *auto*
have *nothing-on-v1*:
 $(\forall c. \|len\ v1\ ts\ c\| = 0$
 $\vee (restrict\ v1\ (clm\ ts)\ c = \emptyset \wedge restrict\ v1\ (res\ ts)\ c = \emptyset))$
by (*metis (no-types, lifting) asm len-empty-on-subview1 non-empty-v1-v2*
restriction-stable1)
have *nothing-on-v2*:
 $(\forall c. \|len\ v2\ ts\ c\| = 0$
 $\vee (restrict\ v2\ (clm\ ts)\ c = \emptyset \wedge restrict\ v2\ (res\ ts)\ c = \emptyset))$
by (*metis (no-types, lifting) asm len-empty-on-subview2 non-empty-v1-v2*
restriction-stable2)
have
 $(v=v1 \parallel v2)$
 $\wedge 0 < \|ext\ v1\| \wedge |lan\ v1| = 1$
 $\wedge (\forall c. \|len\ v1\ ts\ c\| = 0$
 $\vee (restrict\ v1\ (clm\ ts)\ c = \emptyset \wedge restrict\ v1\ (res\ ts)\ c = \emptyset))$
 $\wedge 0 < \|ext\ v2\| \wedge |lan\ v2| = 1$
 $\wedge (\forall c. \|len\ v2\ ts\ c\| = 0$
 $\vee (restrict\ v2\ (clm\ ts)\ c = \emptyset \wedge restrict\ v2\ (res\ ts)\ c = \emptyset))$
using *non-empty-v1-v2 nothing-on-v1 nothing-on-v2 one-lane* **by** *blast*
then show *ts,v* $\models (free \frown free)$ **by** *blast*
qed

lemma *free-compose*: $\models (free \frown free) \rightarrow free$
proof (*rule allI|rule impI*)
fix *ts v*
assume *asm:ts,v* $\models free \frown free$
have *len-ge-0*: $\|ext\ v\| > 0$
using *asm length-meld* **by** *blast*
have *widt-one*: $|lan\ v| = 1$ **using** *asm*

```

  by (metis horizontal-chop-width-stable)
have no-car:
  (∀ c. ||len v ts c|| = 0 ∨ restrict v (clm ts) c = ∅ ∧ restrict v (res ts) c = ∅)
proof (rule ccontr)
  assume
    ¬(∀ c. ||len v ts c|| = 0
      ∨ (restrict v (clm ts) c = ∅ ∧ restrict v (res ts) c = ∅))
  then obtain c
    where ex:
      ||len v ts c|| ≠ 0 ∧ (restrict v (clm ts) c ≠ ∅ ∨ restrict v (res ts) c ≠ ∅)
    by blast
  from ex have 1: ||len v ts c|| > 0
    using less-eq-real-def real-int.length-ge-zero by auto
  have (restrict v (clm ts) c ≠ ∅ ∨ restrict v (res ts) c ≠ ∅) using ex ..
  then show False
proof
  assume restrict v (clm ts) c ≠ ∅
  then show False
    by (metis (no-types, opaque-lifting) assm add.left-neutral ex len-hchop-add
      restriction.restrict-def view.hchop-def)
next
  assume restrict v (res ts) c ≠ ∅
  then show False
    by (metis (no-types, opaque-lifting) assm add.left-neutral ex len-hchop-add
      restriction.restrict-def view.hchop-def)
qed
qed
show ts,v |=free
  using len-ge-0 widt-one no-car by blast
qed

```

```

lemma free-dense: |=free ↔ (free ◊ free)
  using free-decompose free-compose by blast

```

```

lemma free-dense2: |=free → ⊤ ◊ free ◊ ⊤
  using horizontal-chop-empty-left horizontal-chop-empty-right by fastforce

```

The next lemmas show the connection between the spatial. In particular, if the view consists of one lane and a non-zero extension, where neither a reservation nor a car resides, the view satisfies free (and vice versa).

```

lemma no-cars-means-free:
  |=((l>0) ∧ (ω = 1) ∧ (∀ c. ¬ (⊤ ◊ (cl(c) ∨ re(c)) ◊ ⊤))) → free
proof (rule allI|rule impI)+
  fix ts v
  assume assm:
    ts,v |= ((l>0) ∧ (ω = 1) ∧ (∀ c. ¬ (⊤ ◊ (cl(c) ∨ re(c)) ◊ ⊤)))
  have ge-0: ts,v |= l > 0 using assm by best
  have one-lane: ts,v |= ω = 1 using assm by best

```

show $ts, v \models \text{free}$
proof (*rule ccontr*)
have *no-car*: $ts, v \models \neg(\exists c. (\top \frown (cl(c) \vee re(c)) \frown \top))$
using *asm* **by** *best*
assume $ts, v \models \neg \text{free}$
hence *contra*:
 $\neg(\forall c. \|len\ v\ ts\ c\| = 0 \vee \text{restrict } v\ (clm\ ts)\ c = \emptyset \wedge \text{restrict } v\ (res\ ts)\ c = \emptyset)$
using *ge-0 one-lane* **by** *blast*
hence *ex-car*:
 $\exists c. \|len\ v\ ts\ c\| > 0 \wedge (\text{restrict } v\ (clm\ ts)\ c \neq \emptyset \vee \text{restrict } v\ (res\ ts)\ c \neq \emptyset)$
using *real-int.length-ge-zero dual-order.antisym not-le*
by *metis*
obtain *c* **where** *c-def*:
 $\|len\ v\ ts\ c\| > 0 \wedge (\text{restrict } v\ (clm\ ts)\ c \neq \emptyset \vee \text{restrict } v\ (res\ ts)\ c \neq \emptyset)$
using *ex-car* **by** *blast*
hence $(\text{restrict } v\ (clm\ ts)\ c \neq \emptyset \vee \text{restrict } v\ (res\ ts)\ c \neq \emptyset)$ **by** *best*
thus *False*
proof
assume $\text{restrict } v\ (clm\ ts)\ c \neq \emptyset$
with *one-lane* **have** *clm-one*: $|\text{restrict } v\ (clm\ ts)\ c| = 1$
using *el-in-restriction-clm-singleton*
by (*metis card-non-empty-geq-one dual-order.antisym restriction.restriction-clm-leq-one*)
obtain *v1* **and** *v2* **and** *v3* **and** *v4*
where $v = v1 \parallel v2$ **and** $v2 = v3 \parallel v4$
and *len-eq*: $len\ v3\ ts\ c = ext\ v3 \wedge \|len\ v3\ ts\ c\| = \|len\ v\ ts\ c\|$
using *horizontal-chop-empty-left horizontal-chop-empty-right*
len-fills-subview c-def **by** *blast*
then **have** *res-non-empty*: $\text{restrict } v3\ (clm\ ts)\ c \neq \emptyset$
using $\langle \text{restrict } v\ (clm\ ts)\ c \neq \emptyset \rangle$ *restriction-stable restriction-stable1*
by *auto*
have *len-non-empty*: $\|len\ v3\ ts\ c\| > 0$
using *len-eq c-def* **by** *auto*
have $|\text{restrict } v3\ (clm\ ts)\ c| = 1$
using $\langle v2 = v3 \parallel v4 \rangle \langle v = v1 \parallel v2 \rangle$ *clm-one restriction-stable restriction-stable1*
by *auto*
have *v3-one-lane*: $|lan\ v3| = 1$
using $\langle v2 = v3 \parallel v4 \rangle \langle v = v1 \parallel v2 \rangle$ *hchop-def one-lane*
by *auto*
have *clm-fills-v3*: $\text{restrict } v3\ (clm\ ts)\ c = lan\ v3$
proof (*rule ccontr*)
assume *aux*: $\text{restrict } v3\ (clm\ ts)\ c \neq lan\ v3$
have $\text{restrict } v3\ (clm\ ts)\ c \sqsubseteq lan\ v3$
by (*simp add: restrict-view*)
hence $\exists n. n \notin \text{restrict } v3\ (clm\ ts)\ c \wedge n \in lan\ v3$
using $\langle \text{restrict } v3\ (clm\ ts)\ c \neq lan\ v3 \rangle$
restriction.restrict-eq-lan-subs v3-one-lane
by *auto*
hence $|lan\ v3| > 1$
using $\langle |(\text{restrict } v3\ (clm\ ts)\ c)| = 1 \rangle \langle \text{restrict } v3\ (clm\ ts)\ c \leq lan\ v3 \rangle$ *aux*

```

      restriction.restrict-eq-lan-subst v3-one-lane
    by auto
  thus False using v3-one-lane by auto
qed
have  $\|ext\ v3\| > 0$  using c-def len-eq by auto
have  $ts, v3 \models cl(c)$  using clm-one len-eq c-def clm-fills-v3 v3-one-lane
  by auto
hence  $ts, v \models (\top \frown (cl(c) \vee re(c)) \frown \top)$ 
  using  $\langle v2=v3\|v4 \rangle \langle v=v1\|v2 \rangle$  by blast
hence  $ts, v \models \exists c. (\top \frown (cl(c) \vee re(c)) \frown \top)$  by blast
thus False using no-car by best
next
assume restrict v (res ts) c  $\neq \emptyset$ 
with one-lane have clm-one:|restrict v (res ts) c| = 1
  using el-in-restriction-clm-singleton
  by (metis nat-int.card-non-empty-geq-one nat-int.card-subset-le
    dual-order.antisym restrict-view)
obtain v1 and v2 and v3 and v4
  where  $v=v1\|v2$  and  $v2=v3\|v4$ 
    and len-eq:len v3 ts c = ext v3  $\wedge$   $\|len\ v3\ ts\ c\| = \|len\ v\ ts\ c\|$ 
  using horizontal-chop-empty-left horizontal-chop-empty-right
    len-fills-subview c-def by blast
then have res-non-empty:restrict v3 (res ts) c  $\neq \emptyset$ 
  using  $\langle restrict\ v\ (res\ ts)\ c \neq \emptyset \rangle$  restriction-stable restriction-stable1
  by auto
have len-non-empty: $\|len\ v3\ ts\ c\| > 0$ 
  using len-eq c-def by auto
have |restrict v3 (res ts) c| = 1
  using  $\langle v2=v3\|v4 \rangle \langle v=v1\|v2 \rangle$  clm-one restriction-stable restriction-stable1
  by auto
have v3-one-lane:|lan v3| = 1
  using  $\langle v2=v3\|v4 \rangle \langle v=v1\|v2 \rangle$  hchop-def one-lane
  by auto
have restrict v3 (res ts) c = lan v3
proof (rule ccontr)
  assume aux:restrict v3 (res ts) c  $\neq$  lan v3
  have restrict v3 (res ts) c  $\sqsubseteq$  lan v3
    by (simp add: restrict-view)
  hence  $\exists n. n \notin restrict\ v3\ (res\ ts)\ c \wedge n \in lan\ v3$ 
    using aux  $\langle |restrict\ v3\ (res\ ts)\ c| = 1 \rangle$  restriction.restrict-eq-lan-subst
v3-one-lane
    by auto
  hence |lan v3| > 1
    using  $\langle |restrict\ v3\ (res\ ts)\ c| = 1 \rangle \langle restrict\ v3\ (res\ ts)\ c \leq lan\ v3 \rangle$  aux
    restriction.restrict-eq-lan-subst v3-one-lane
    by auto
  thus False using v3-one-lane by auto
qed
have  $\|ext\ v3\| > 0$  using c-def len-eq by auto

```

have $ts, v3 \models re(c)$
using *clm-one len-eq c-def* $\langle restrict\ v3\ (res\ ts)\ c = lan\ v3 \rangle$ *v3-one-lane*
by *auto*
hence $ts, v \models (\top \frown (cl(c) \vee re(c)) \frown \top)$
using $\langle v2=v3 \parallel v4 \rangle \langle v=v1 \parallel v2 \rangle$ **by** *blast*
hence $ts, v \models \exists c. (\top \frown (cl(c) \vee re(c)) \frown \top)$ **by** *blast*
thus *False* **using** *no-car* **by** *best*
qed
qed
qed

lemma *free-means-no-cars*:

$\models free \rightarrow ((\mathbf{1} > 0) \wedge (\omega = 1) \wedge (\forall c. \neg (\top \frown (cl(c) \vee re(c)) \frown \top)))$

proof (*rule allI* | *rule impI*)**+**

fix $ts\ v$

assume $assm: ts, v \models free$

have $no-car: ts, v \models (\forall c. \neg (\top \frown (cl(c) \vee re(c)) \frown \top))$

proof (*rule ccontr*)

assume $\neg (ts, v \models (\forall c. \neg (\top \frown (cl(c) \vee re(c)) \frown \top)))$

hence $contra: ts, v \models \exists c. \top \frown (cl(c) \vee re(c)) \frown \top$ **by** *blast*

from *this* **obtain** c **and** $v1$ **and** v' **and** $v2$ **and** vc **where**

$vc-def: (v=v1 \parallel v') \wedge (v'=vc \parallel v2) \wedge (ts, vc \models cl(c) \vee re(c))$ **by** *blast*

hence $len-ge-zero: \|len\ v\ ts\ c\| > 0$

by (*metis len-empty-on-subview1 len-empty-on-subview2 less-eq-real-def real-int.length-ge-zero*)

from $vc-def$ **have** $vc-ex-car$:

$restrict\ vc\ (clm\ ts)\ c \neq \emptyset \vee restrict\ vc\ (res\ ts)\ c \neq \emptyset$

using *nat-int.card-empty-zero one-neq-zero* **by** *auto*

have $eq-lan: lan\ v = lan\ vc$ **using** *vc-def hchop-def* **by** *auto*

hence $v-ex-car: restrict\ v\ (clm\ ts)\ c \neq \emptyset \vee restrict\ v\ (res\ ts)\ c \neq \emptyset$

using $vc-ex-car$ **by** (*simp add: restrict-def*)

from $len-ge-zero$ **and** $v-ex-car$ **and** $assm$ **show** *False* **by** *force*

qed

with $assm$ **show**

$ts, v \models ((\mathbf{1} > 0) \wedge (\omega = 1) \wedge (\forall c. \neg (\top \frown (cl(c) \vee re(c)) \frown \top)))$

by *blast*

qed

lemma *free-eq-no-cars*:

$\models free \leftrightarrow ((\mathbf{1} > 0) \wedge (\omega = 1) \wedge (\forall c. \neg (\top \frown (cl(c) \vee re(c)) \frown \top)))$

using *no-cars-means-free free-means-no-cars* **by** *blast*

lemma *free-nowhere-res*: $\models free \rightarrow \neg (\top \frown (re(c)) \frown \top)$

using *free-eq-no-cars* **by** *blast*

lemma *two-res-not-res*: $\models ((re(c) \smile re(c)) \rightarrow \neg re(c))$

by (*metis add-eq-self-zero one-neq-zero width-add1*)

lemma *two-clm-width*: $\models ((cl(c) \smile cl(c)) \rightarrow \omega = 2)$


```

    by (metis one-add-one width-add1)

lemma two-res-no-car:  $\models (re(c) \smile re(c)) \rightarrow \neg(\exists c. (cl(c) \vee re(c)))$ 
  by (metis add-eq-self-zero one-neq-zero width-add1)

lemma two-lanes-no-car:  $\models (\neg \omega = 1) \rightarrow \neg(\exists c. (cl(c) \vee re(c)))$ 
  by simp

lemma empty-no-car:  $\models (1 = 0) \rightarrow \neg(\exists c. (cl(c) \vee re(c)))$ 
  by simp

lemma car-one-lane-non-empty:  $\models (\exists c. (cl(c) \vee re(c))) \rightarrow ((\omega = 1) \wedge (1 > 0))$ 
  by blast

lemma one-lane-notfree:
   $\models (\omega = 1) \wedge (1 > 0) \wedge (\neg free) \rightarrow ((\top \frown (\exists c. (re(c) \vee cl(c))) \frown \top))$ 
proof (rule allI|rule impI)+
  fix ts v
  assume assm:  $ts, v \models (\omega = 1) \wedge (1 > 0) \wedge (\neg free)$ 
  hence not-free:  $ts, v \models \neg free$  by blast
  with free-eq-no-cars have
     $ts, v \models \neg ((1 > 0) \wedge (\omega = 1) \wedge (\forall c. \neg (\top \frown (cl(c) \vee re(c)) \frown \top)))$ 
  by blast
  hence  $ts, v \models \neg (\forall c. \neg (\top \frown (cl(c) \vee re(c)) \frown \top))$ 
  using assm by blast
  thus  $ts, v \models (\top \frown (\exists c. (re(c) \vee cl(c))) \frown \top)$  by blast
qed

lemma one-lane-empty-or-car:
   $\models (\omega = 1) \wedge (1 > 0) \rightarrow (free \vee (\top \frown (\exists c. (re(c) \vee cl(c))) \frown \top))$ 
  using one-lane-notfree by blast
end
end

```

11 Perfect Sensors

This section contains an instantiations of the sensor function for "perfect sensors". That is, each car can perceive both the physical size as well as the braking distance of each other car.

```

theory Perfect-Sensors
  imports ../Length
begin

```

```

definition perfect::cars  $\Rightarrow$  traffic  $\Rightarrow$  cars  $\Rightarrow$  real
  where perfect e ts c  $\equiv$  traffic.physical-size ts c + traffic.braking-distance ts c

```

```

locale perfect-sensors = traffic+view
begin

```

interpretation *perfect-sensors* : *sensors perfect* :: *cars* \Rightarrow *traffic* \Rightarrow *cars* \Rightarrow *real*
proof *unfold-locales*
fix *e ts c*
show $0 < \text{perfect } e \text{ } ts \text{ } c$
by (*metis less-add-same-cancel2 less-trans perfect-def traffic.psGeZero traf-*
fic.sdGeZero)
qed

notation *perfect-sensors.space* (*space*)
notation *perfect-sensors.len* (*len*)

With this sensor definition, we can show that the perceived length of a car is independent of the spatial transitions between traffic snapshots. The length may only change during evolutions, in particular if the car changes its dynamical behaviour.

lemma *create-reservation-length-stable*:
 $(ts - r(d) \rightarrow ts') \longrightarrow \text{len } v \text{ } ts \text{ } c = \text{len } v \text{ } ts' \text{ } c$

proof
assume *assm:(ts-r(d)→ts')*
hence *eq:space ts v c = space ts' v c*
using *traffic.create-reservation-def perfect-sensors.space-def perfect-def*
by (*simp*)
show $\text{len } v \text{ } (ts) \text{ } c = \text{len } v \text{ } (ts') \text{ } c$
proof (*cases left ((space ts v) c) > right (ext v)*)
assume *outside-right:left ((space ts v) c) > right (ext v)*
hence *outside-right':left ((space ts' v) c) > right (ext v)* **using** *eq by simp*
from *outside-right* **and** *outside-right'* **show** *?thesis*
by (*simp add: perfect-sensors.len-def eq*)
next
assume *inside-right:¬ left ((space ts v) c) > right (ext v)*
hence *inside-right':¬ left ((space ts' v) c) > right (ext v)* **using** *eq by simp*
show $\text{len } v \text{ } (ts) \text{ } c = \text{len } v \text{ } (ts') \text{ } c$
proof (*cases left (ext v) > right ((space ts v) c)*)
assume *outside-left:left (ext v) > right ((space ts v) c)*
hence *outside-left':left (ext v) > right ((space ts' v) c)* **using** *eq by simp*
from *outside-left* **and** *outside-left'* **show** *?thesis*
by (*simp add: perfect-sensors.len-def eq*)
next
assume *inside-left:¬ left (ext v) > right ((space ts v) c)*
hence *inside-left':¬ left (ext v) > right ((space ts' v) c)* **using** *eq by simp*
from *inside-left inside-right inside-left' inside-right' eq*
show *?thesis by (simp add: perfect-sensors.len-def)*
qed
qed
qed

lemma *create-claim-length-stable*:
 $(ts - c(d,n) \rightarrow ts') \longrightarrow \text{len } v \text{ } ts \text{ } c = \text{len } v \text{ } ts' \text{ } c$

proof
assume $assm:(ts-c(d,n)\rightarrow ts')$
hence $eq:space\ ts\ v\ c = space\ ts'\ v\ c$
using $traffic.create-claim-def\ perfect-sensors.space-def\ perfect-def$
by $(simp)$
show $len\ v\ (ts)\ c = len\ v\ (ts')\ c$
proof $(cases\ left\ ((space\ ts\ v)\ c) > right\ (ext\ v))$
assume $outside-right:left\ ((space\ ts\ v)\ c) > right\ (ext\ v)$
hence $outside-right':left\ ((space\ ts'\ v)\ c) > right\ (ext\ v)$ **using** eq **by** $simp$
from $outside-right$ **and** $outside-right'$ **show** $?thesis$
by $(simp\ add:\ perfect-sensors.len-def\ eq)$
next
assume $inside-right:\neg\ left\ ((space\ ts\ v)\ c) > right\ (ext\ v)$
hence $inside-right':\neg\ left\ ((space\ ts'\ v)\ c) > right\ (ext\ v)$ **using** eq **by** $simp$
show $len\ v\ (ts)\ c = len\ v\ (ts')\ c$
proof $(cases\ left\ (ext\ v) > right\ ((space\ ts\ v)\ c))$
assume $outside-left:left\ (ext\ v) > right\ ((space\ ts\ v)\ c)$
hence $outside-left':left\ (ext\ v) > right\ ((space\ ts'\ v)\ c)$ **using** eq **by** $simp$
from $outside-left$ **and** $outside-left'$ **show** $?thesis$
by $(simp\ add:\ perfect-sensors.len-def\ eq)$
next
assume $inside-left:\neg\ left\ (ext\ v) > right\ ((space\ ts\ v)\ c)$
hence $inside-left':\neg\ left\ (ext\ v) > right\ ((space\ ts'\ v)\ c)$ **using** eq **by** $simp$
from $inside-left\ inside-right\ inside-left'\ inside-right'\ eq$
show $?thesis$ **by** $(simp\ add:\ perfect-sensors.len-def)$
qed
qed
qed

lemma $withdraw-reservation-length-stable:$

$(ts-wdr(d,n)\rightarrow ts') \longrightarrow len\ v\ ts\ c = len\ v\ ts'\ c$

proof
assume $assm:(ts-wdr(d,n)\rightarrow ts')$
hence $eq:space\ ts\ v\ c = space\ ts'\ v\ c$
using $traffic.withdraw-reservation-def\ perfect-sensors.space-def\ perfect-def$
by $(simp)$
show $len\ v\ (ts)\ c = len\ v\ (ts')\ c$
proof $(cases\ left\ ((space\ ts\ v)\ c) > right\ (ext\ v))$
assume $outside-right:left\ ((space\ ts\ v)\ c) > right\ (ext\ v)$
hence $outside-right':left\ ((space\ ts'\ v)\ c) > right\ (ext\ v)$ **using** eq **by** $simp$
from $outside-right$ **and** $outside-right'$ **show** $?thesis$
by $(simp\ add:\ perfect-sensors.len-def\ eq)$
next
assume $inside-right:\neg\ left\ ((space\ ts\ v)\ c) > right\ (ext\ v)$
hence $inside-right':\neg\ left\ ((space\ ts'\ v)\ c) > right\ (ext\ v)$ **using** eq **by** $simp$
show $len\ v\ (ts)\ c = len\ v\ (ts')\ c$
proof $(cases\ left\ (ext\ v) > right\ ((space\ ts\ v)\ c))$
assume $outside-left:left\ (ext\ v) > right\ ((space\ ts\ v)\ c)$
hence $outside-left':left\ (ext\ v) > right\ ((space\ ts'\ v)\ c)$ **using** eq **by** $simp$

```

    from outside-left and outside-left' show ?thesis
    by (simp add: perfect-sensors.len-def eq)
next
  assume inside-left: $\neg$  left (ext v) > right ((space ts v) c)
  hence inside-left': $\neg$  left (ext v) > right ((space ts' v) c) using eq by simp
  from inside-left inside-right inside-left' inside-right' eq
  show ?thesis by (simp add: perfect-sensors.len-def)
qed
qed
qed

lemma withdraw-claim-length-stable:
  (ts-wdc(d) $\rightarrow$ ts')  $\longrightarrow$  len v ts c = len v ts' c
proof
  assume assm:(ts-wdc(d) $\rightarrow$ ts')
  hence eq:space ts v c = space ts' v c
    using traffic.withdraw-claim-def perfect-sensors.space-def perfect-def
    by (simp)
  show len v ( ts ) c = len v ( ts' ) c
proof (cases left ((space ts v) c) > right (ext v))
  assume outside-right:left ((space ts v) c) > right (ext v)
  hence outside-right':left ((space ts' v) c) > right (ext v) using eq by simp
  from outside-right and outside-right' show ?thesis
    by (simp add: perfect-sensors.len-def eq)
next
  assume inside-right: $\neg$  left ((space ts v) c) > right (ext v)
  hence inside-right': $\neg$  left ((space ts' v) c) > right (ext v) using eq by simp
  show len v ( ts ) c = len v ( ts' ) c
proof (cases left (ext v) > right ((space ts v) c) )
  assume outside-left: left (ext v) > right ((space ts v) c)
  hence outside-left': left (ext v) > right ((space ts' v) c) using eq by simp
  from outside-left and outside-left' show ?thesis
    by (simp add: perfect-sensors.len-def eq)
next
  assume inside-left: $\neg$  left (ext v) > right ((space ts v) c)
  hence inside-left': $\neg$  left (ext v) > right ((space ts' v) c) using eq by simp
  from inside-left inside-right inside-left' inside-right' eq
  show ?thesis by (simp add: perfect-sensors.len-def)
qed
qed
qed

```

The following lemma shows that the perceived length is independent from the owner of the view. That is, as long as two views consist of the same extension, the perceived length of each car is the same in both views.

```

lemma all-own-ext-eq-len-eq:
  ext v = ext v'  $\longrightarrow$  len v ts c = len v' ts c
proof
  assume assm:ext v = ext v'

```

```

hence sp:space ts v c = space ts v' c
  by (simp add: perfect-def perfect-sensors.space-def)
have left-eq:left (ext v) = left (ext v') using asm by simp
have right-eq:right (ext v) = right (ext v') using asm by simp
show len v ( ts) c = len v' ( ts) c
proof (cases left ((space ts v) c) > right (ext v))
  assume outside-right:left ((space ts v) c) > right (ext v)
  hence outside-right':left ((space ts v) c) > right (ext v')
    using right-eq by simp
  from outside-right and outside-right' show ?thesis
    by (simp add: perfect-sensors.len-def right-eq asm sp)
next
assume inside-right:¬ left ((space ts v) c) > right (ext v)
hence inside-right':¬ left ((space ts v) c) > right (ext v')
  using right-eq by simp
show len v ( ts) c = len v' ( ts) c
proof (cases left (ext v) > right ((space ts v) c))
  assume outside-left:left (ext v) > right ((space ts v) c)
  hence outside-left':left (ext v') > right ((space ts v) c)
    using left-eq by simp
  from outside-left and outside-left' show ?thesis
    using perfect-sensors.len-def left-eq sp right-eq
    by auto
next
assume inside-left:¬ left (ext v) > right ((space ts v) c)
hence inside-left':¬ left (ext v') > right ((space ts v) c)
  using left-eq by simp
  from inside-left inside-right inside-left' inside-right' left-eq right-eq
  show ?thesis by (simp add:perfect-sensors.len-def sp)
qed
qed
qed

```

Finally, switching the perspective of a view does not change the perceived length.

```

lemma switch-length-stable:(v=d>v') → len v ts c = len v' ts c
  using all-own-ext-eq-len-eq view.switch-def by metis

```

```

end
end

```

12 HMLSL for Perfect Sensors

Within this section, we instantiate HMLSL for cars with perfect sensors.

```

theory HMLSL-Perfect
  imports ../HMLSL Perfect-Sensors
begin

```

locale *hmlsl-perfect* = *perfect-sensors* + *restriction*
begin

interpretation *hmlsl* : *hmlsl perfect* :: *cars* \Rightarrow *traffic* \Rightarrow *cars* \Rightarrow *real*
proof *unfold-locales*

fix *e ts c*
show $0 < \text{perfect } e \text{ ts } c$
by (*metis less-add-same-cancel2 less-trans perfect-def traffic.psGeZero traffic.sdGeZero*)
qed

notation *hmlsl.re* (*re*'(-'))
notation *hmlsl.cl* (*cl*'(-'))
notation *hmlsl.len* (*len*)

The spatial atoms are independent of the perspective of the view. Hence we can prove several lemmas on the relation between the hybrid modality and the spatial atoms.

lemma *at-res1*: $\models (re(c)) \rightarrow (\forall d. @d \text{ re}(c))$
by (*metis (no-types, lifting) perfect-sensors.switch-length-stable restriction.switch-restrict-stable view.switch-def*)

lemma *at-res2*: $\models (\forall d. @d \text{ re}(c)) \rightarrow re(c)$
using *view.switch-refl* **by** *blast*

lemma *at-res*: $\models re(c) \leftrightarrow (\forall d. @d \text{ re}(c))$
using *at-res1 at-res2* **by** *blast*

lemma *at-res-inst*: $\models (@d \text{ re}(c)) \rightarrow re(c)$
proof (*rule allI|rule impI*)+

fix *ts v*
assume *assm*: *ts, v* $\models (@d \text{ re}(c))$
obtain *v'* **where** *v'-def*: $(v=(d) > v')$
using *view.switch-always-exists* **by** *blast*
with *assm* **have** *v':ts, v'* $\models re(c)$ **by** *blast*
with *v'* **show** *ts, v* $\models re(c)$
using *restriction.switch-restrict-stable perfect-sensors.switch-length-stable v'-def view.switch-def*
by (*metis (no-types, lifting) all-own-ext-eq-len-eq*)
qed

lemma *at-clm1*: $\models cl(c) \rightarrow (\forall d. @d \text{ cl}(c))$
by (*metis (no-types, lifting) all-own-ext-eq-len-eq view.switch-def restriction.switch-restrict-stable*)

lemma *at-clm2*: $\models (\forall d. @d \text{ cl}(c)) \rightarrow cl(c)$
using *view.switch-def* **by** *auto*

```

lemma at-clm: $\models cl(c) \leftrightarrow (\forall d. @d cl(c))$ 
  using at-clm1 at-clm2 by blast

lemma at-clm-inst: $\models (@d cl(c)) \rightarrow cl(c)$ 
proof (rule allI|rule impI)+
  fix ts v
  assume assm: $ts, v \models (@d cl(c))$ 
  obtain v' where v'-def: $(v=(d) > v')$ 
    using view.switch-always-exists by blast
  with assm have v':ts, v' \models cl(c) by blast
  with v' show  $ts, v \models cl(c)$ 
    using restriction.switch-restrict-stable switch-length-stable v'-def view.switch-def

  by (metis (no-types, lifting) all-own-ext-eq-len-eq)
qed

```

With the definition of sensors, we can also express how the spatial situation changes after the different transitions. In particular, we can prove lemmas corresponding to the activity and stability rules of the proof system for MLSL [5].

Observe that we were not able to prove these rules for basic HMLS, since its generic sensor function allows for instantiations where the perceived length changes during spatial transitions.

```

lemma backwards-res-act:
   $(ts \dashv r(c) \rightarrow ts') \wedge (ts', v \models re(c)) \longrightarrow (ts, v \models re(c) \vee cl(c))$ 
proof
  assume assm: $(ts \dashv r(c) \rightarrow ts') \wedge (ts', v \models re(c))$ 
  from assm have len-eq: $len v ts c = len v ts' c$ 
    using create-reservation-length-stable by blast
  have res ts c \sqsubseteq res ts' c using assm traffic.create-res-subseteq1 by blast
  hence restr-sub-res: $restrict v (res ts) c \sqsubseteq restrict v (res ts') c$ 
    by (simp add: restriction.restrict-view assm)
  have clm ts c \sqsubseteq res ts' c using assm traffic.create-res-subseteq2 by blast
  hence restr-sub-clm: $restrict v (clm ts) c \sqsubseteq restrict v (res ts') c$ 
    by (simp add: restriction.restrict-view assm)
  have  $restrict v (res ts) c = \emptyset \vee restrict v (res ts) c \neq \emptyset$  by simp
  then show  $ts, v \models (re(c) \vee cl(c))$ 
proof
  assume restr-res-nonempty: $restrict v (res ts) c \neq \emptyset$ 
  hence restrict-one: $|restrict v (res ts) c| = 1$ 
  using nat-int.card-non-empty-geq-one nat-int.card-subset-le dual-order.antisym
    restr-sub-res assm by fastforce
  have  $restrict v (res ts) c \sqsubseteq lan v$ 
    using restr-sub-res assm by auto
  hence  $restrict v (res ts) c = lan v$ 
    using restriction.restrict-eq-lan-sub restrict-one assm by auto
  thus  $ts, v \models (re(c) \vee cl(c))$ 

```

```

    using assm len-eq by auto
next
assume restr-res-empty: restrict v (res ts) c =  $\emptyset$ 
then have clm-non-empty: restrict v (clm ts) c  $\neq \emptyset$ 
  by (metis assm inter-empty2 local.hmlsl.free-no-clm
      restriction.create-reservation-restrict-union restriction.restrict-def'
      un-empty-absorb1)
hence restrict-one: |restrict v (clm ts) c| = 1
using nat-int.card-non-empty-geq-one nat-int.card-subset-le dual-order.antisym
  restr-subst-clm assm by fastforce
have restrict v (clm ts) c  $\sqsubseteq$  lan v
  using restr-subst-clm assm by auto
hence restrict v (clm ts) c = lan v
  using restriction.restrict-eq-lan-subst restrict-one assm by auto
thus  $ts, v \models (re(c) \vee cl(c))$ 
  using assm len-eq by auto
qed
qed

```

lemma *backwards-res-act-somewhere*:
 $(ts \text{--}r(c) \rightarrow ts') \wedge (ts', v \models \langle re(c) \rangle) \longrightarrow (ts, v \models \langle re(c) \vee cl(c) \rangle)$
 using *backwards-res-act* by *blast*

lemma *backwards-res-stab*:
 $(ts \text{--}r(d) \rightarrow ts') \wedge (d \neq c) \wedge (ts', v \models re(c)) \longrightarrow (ts, v \models re(c))$
 using *perfect-sensors.create-reservation-length-stable restriction.restrict-def'*
traffic.create-res-subseteq1-neq
 by *auto*

lemma *backwards-c-res-stab*:
 $(ts \text{--}c(d, n) \rightarrow ts') \wedge (ts', v \models re(c)) \longrightarrow (ts, v \models re(c))$
 using *create-claim-length-stable traffic.create-clm-eq-res*
 by (*metis (mono-tags, lifting) traffic.create-claim-def*)

lemma *backwards-wdc-res-stab*:
 $(ts \text{--}wdc(d) \rightarrow ts') \wedge (ts', v \models re(c)) \longrightarrow (ts, v \models re(c))$
 using *withdraw-claim-length-stable traffic.withdraw-clm-eq-res*
 by (*metis (mono-tags, lifting) traffic.withdraw-claim-def*)

lemma *backwards-wdr-res-stab*:
 $(ts \text{--}wdr(d, n) \rightarrow ts') \wedge (ts', v \models re(c)) \longrightarrow (ts, v \models re(c))$
 by (*metis inf.absorb1 order-trans perfect-sensors.withdraw-reservation-length-stable*
restriction.restrict-def' restriction.restrict-res traffic.withdraw-res-subseteq)

We now proceed to prove the *reservation lemma*, which was crucial in the manual safety proof [2].

lemma *reservation1*: $\models (re(c) \vee cl(c)) \rightarrow \Box r(c) re(c)$
proof (*rule allI* | *rule impI*) +
 fix $ts \ v \ ts'$


```

assume assm: $ts, v \models re(c) \vee cl(c)$  and ts'-def: $ts - r(c) \rightarrow ts'$ 
from assm show  $ts', v \models re(c)$ 
proof
  assume  $re:ts, v \models re(c)$ 
  show ?thesis
  by (metis inf.absorb1 order-trans perfect-sensors.create-reservation-length-stable
    re restriction.restrict-def' restriction.restrict-subseteq
    traffic.create-res-subseteq1 ts'-def)
next
  assume  $cl:ts, v \models cl(c)$ 
  show ?thesis
  by (metis cl inf.absorb1 order-trans perfect-sensors.create-reservation-length-stable
    restriction.restrict-def' restriction.restrict-subseteq
    traffic.create-res-subseteq2 ts'-def)
qed
qed

lemma reservation2:  $\models(\Box r(c) re(c)) \rightarrow (re(c) \vee cl(c))$ 
  using backwards-res-act traffic.always-create-res by blast

lemma reservation:  $\models(\Box r(c) re(c)) \leftrightarrow (re(c) \vee cl(c))$ 
  using reservation1 reservation2 by blast
end
end

```

13 Safety for Cars with Perfect Sensors

This section contains the definition of requirements for lane change and distance controllers for cars, with the assumption of perfect sensors. Using these definitions, we show that safety is an invariant along all possible behaviour of cars.

```

theory Safety-Perfect
  imports HMLSL-Perfect
begin

context hmlsl-perfect
begin
interpretation hmlsl : hmlsl perfect :: cars  $\Rightarrow$  traffic  $\Rightarrow$  cars  $\Rightarrow$  real
proof unfold-locales
  fix e ts c
  show  $0 < perfect\ e\ ts\ c$ 
  by (metis less-add-same-cancel2 less-trans perfect-def traffic.psGeZero
    traffic.sdGeZero)
qed

notation hmlsl.re (re'(-))
notation hmlsl.cl (cl'(-))
notation hmlsl.len (len)

```

Safety in the context of HMLSL means the absence of overlapping reservations. Using the somewhere modality, this is easy to formalise.

abbreviation $safe::cars \Rightarrow \sigma$
where $safe\ e \equiv \forall\ c. \neg(c = e) \rightarrow \neg \langle re(c) \wedge re(e) \rangle$

The distance controller ensures, that as long as the cars do not try to change their lane, they keep their distance. More formally, if the reservations of two cars do not overlap, they will also not overlap after an arbitrary amount of time passed. Observe that the cars are allowed to change their dynamical behaviour, i.e., to accelerate and brake.

abbreviation $DC::\sigma$
where $DC \equiv \mathbf{G}(\forall\ c\ d. \neg(c = d) \rightarrow \neg \langle re(c) \wedge re(d) \rangle) \rightarrow \Box \tau \neg \langle re(c) \wedge re(d) \rangle$

To identify possibly dangerous situations during a lane change manoeuvre, we use the *potential collision check*. It allows us to identify situations, where the claim of a car d overlaps with any part of the car c .

abbreviation $pcc::cars \Rightarrow cars \Rightarrow \sigma$
where $pcc\ c\ d \equiv \neg(c = d) \wedge \langle cl(d) \wedge (re(c) \vee cl(c)) \rangle$

The only restriction the lane change controller imposes onto the cars is that in the case of a potential collision, they are not allowed to change the claim into a reservation.

abbreviation $LC::\sigma$
where $LC \equiv \mathbf{G}(\forall\ d. (\exists\ c. pcc\ c\ d) \rightarrow \Box r(d) \perp)$

The safety theorem is as follows. If the controllers of all cars adhere to the specifications given by LC and DC , and we start with an initially safe traffic snapshot, then all reachable traffic snapshots are also safe.

theorem $safety: \models (\forall\ e. safe\ e) \wedge DC \wedge LC \rightarrow \mathbf{G}(\forall\ e. safe\ e)$

proof (*rule allI* | *rule impI*) +

fix $ts\ v\ ts'$

fix $e\ c::cars$

assume $assm: ts, v \models (\forall\ e. safe\ e) \wedge DC \wedge LC$

assume $abs: ts \Rightarrow ts'$

assume $nequals: ts, v \models \neg(c = e)$

from $assm$ **have** $init: ts, v \models (\forall\ e. safe\ e)$ **by** *simp*

from $assm$ **have** $DC: ts, v \models DC$ **by** *simp*

from $assm$ **have** $LC: ts, v \models LC$ **by** *simp*

from abs **show** $ts', move\ ts\ ts'\ v \models \neg \langle re(c) \wedge re(e) \rangle$

proof (*induction*)

case (*refl*)

have $move\ ts\ ts'\ v = v$ **using** *traffic.move-nothing* **by** *simp*

thus *?case* **using** *init traffic.move-nothing nequals* **by** *auto*

next

case (*evolve* $ts'\ ts''$)

have *local-DC*:
 $ts', \text{move } ts \ ts' \ v \models \forall c \ d. \neg(c = d) \rightarrow \neg\langle re(c) \wedge re(d) \rangle \rightarrow \Box \tau \neg\langle re(c) \wedge re(d) \rangle$
using *evolve.hyps DC* **by** *simp*
show *?case*
proof
assume *e-def*: $(ts'', \text{move } ts \ ts'' \ v \models \langle re(c) \wedge re(e) \rangle)$
from *evolve.IH* **and** *nequals* **have**
 $ts' \text{-safe}: ts', \text{move } ts \ ts' \ v \models \neg(c = e) \wedge \neg\langle re(c) \wedge re(e) \rangle$ **by** *fastforce*
hence *no-coll-after-evol*: $ts', \text{move } ts \ ts' \ v \models \Box \tau \neg\langle re(c) \wedge re(e) \rangle$
using *local-DC* **by** *blast*
have *move-eq*: $\text{move } ts' \ ts'' \ (\text{move } ts \ ts' \ v) = \text{move } ts \ ts'' \ v$
using *evolve.hyps traffic.abstract.evolve traffic.abstract.refl traffic.move-trans*
by *blast*
from *no-coll-after-evol* **and** *evolve.hyps* **have**
 $ts'', \text{move } ts' \ ts'' \ (\text{move } ts \ ts' \ v) \models \neg\langle re(c) \wedge re(e) \rangle$
by *blast*
thus *False* **using** *e-def* **using** *move-eq* **by** *fastforce*
qed
next
case *(cr-res ts' ts'')*
have *local-LC*: $ts', \text{move } ts \ ts' \ v \models (\forall d. (\exists c. pcc \ c \ d) \rightarrow \Box r(d) \perp)$
using *LC cr-res.hyps* **by** *blast*
have $\text{move } ts \ ts' \ v = \text{move } ts' \ ts'' \ (\text{move } ts \ ts' \ v)$
using *traffic.move-stability-res cr-res.hyps traffic.move-trans move-stability-clm* **by** *auto*
hence *move-stab*: $\text{move } ts \ ts' \ v = \text{move } ts \ ts'' \ v$
by *(metis traffic.abstract.simps cr-res.hyps traffic.move-trans)*
show *?case*
proof *(rule)*
assume *e-def*: $(ts'', \text{move } ts \ ts'' \ v \models \langle re(c) \wedge re(e) \rangle)$
obtain *d* **where** *d-def*: $ts' \text{-}r(d) \rightarrow ts''$ **using** *cr-res.hyps* **by** *best*
have $d = e \vee d \neq e$ **by** *simp*
thus *False*
proof
assume *eq*: $d = e$
hence *e-trans*: $ts' \text{-}r(e) \rightarrow ts''$ **using** *d-def* **by** *simp*
from *e-def* **have** $ts'', \text{move } ts \ ts'' \ v \models \langle re(c) \wedge re(e) \rangle$ **by** *auto*
hence $\exists v'. (v' \leq \text{move } ts \ ts'' \ v) \wedge (ts'', v' \models re(c) \wedge re(e))$
using *view.somewhere-leq*
by *meson*
then obtain *v'* **where** *v'-def*:
 $(v' \leq \text{move } ts \ ts'' \ v) \wedge (ts'', v' \models re(c) \wedge re(e))$
by *blast*
with *backwards-res-act* **have** $ts', v' \models re(c) \wedge (re(e) \vee cl(e))$
using *e-def backwards-res-stab nequals*
by *(metis (no-types, lifting) d-def eq)*
hence $\exists v'. (v' \leq \text{move } ts \ ts'' \ v) \wedge (ts', v' \models re(c) \wedge (re(e) \vee cl(e)))$

using v' -def **by** blast
hence $ts',move\ ts\ ts''\ v \models \langle re(c) \wedge (re(e) \vee cl(e)) \rangle$
using view.somewhere-leq **by** meson
hence $ts',move\ ts\ ts''\ v \models \langle re(c) \wedge re(e) \rangle \vee \langle re(c) \wedge cl(e) \rangle$
using hmlsl.somewhere-and-or-distr **by** blast
thus False
proof
assume $assm':ts',move\ ts\ ts''\ v \models \langle re(c) \wedge re(e) \rangle$
have $ts',move\ ts\ ts''\ v \models \neg (c = e)$ **using** nequals **by** blast
thus False **using** $assm'$ cr-res.IH e-def move-stab **by** force
next
assume $assm':ts',move\ ts\ ts''\ v \models \langle re(c) \wedge cl(e) \rangle$
hence $ts',move\ ts\ ts''\ v \models \neg (c = e) \wedge \langle re(c) \wedge cl(e) \rangle$
using e-def nequals **by** force
hence $ts',move\ ts\ ts''\ v \models \neg (c = e) \wedge \langle cl(e) \wedge (re(c) \vee cl(c)) \rangle$ **by** blast
hence $pcc:ts',move\ ts\ ts''\ v \models pcc\ c\ e$ **by** blast
have $ts',move\ ts\ ts''\ v \models (\exists c. pcc\ c\ e) \rightarrow \Box r(e) \perp$
using local-LC move-stab **by** fastforce
hence $ts',move\ ts\ ts''\ v \models \Box r(e) \perp$ **using** pcc **by** blast
thus $ts'',move\ ts\ ts''\ v \models \perp$ **using** e-trans **by** blast
qed
next
assume $neq:d \neq e$
have $c=d \vee c \neq d$ **by** simp
thus False
proof
assume $neq2:c \neq d$
from e-def **have** $ts'',move\ ts\ ts''\ v \models \langle re(c) \wedge re(e) \rangle$ **by** auto
hence $\exists v'. (v' \leq move\ ts\ ts''\ v) \wedge (ts'',v' \models re(c) \wedge re(e))$
using view.somewhere-leq
by meson
then obtain v' **where** v' -def:
 $(v' \leq move\ ts\ ts''\ v) \wedge (ts'',v' \models re(c) \wedge re(e))$
by blast
with backwards-res-stab **have** overlap: $ts',v' \models re(c) \wedge re(e)$
using e-def backwards-res-stab nequals neq2
by (metis (no-types, lifting) d-def neq)
hence $unsafe2:ts',move\ ts\ ts''\ v \models \langle re(c) \wedge re(e) \rangle$
using nequals view.somewhere-leq v' -def **by** blast
from cr-res.IH **have** $ts',move\ ts\ ts''\ v \models \neg \langle re(c) \wedge re(e) \rangle$
using move-stab **by** force
thus False **using** unsafe2 **by** best
next
assume $eq2:c = d$
hence e-trans: $ts' -r(c) \rightarrow ts''$ **using** d-def **by** simp
from e-def **have** $ts'',move\ ts\ ts''\ v \models \langle re(c) \wedge re(e) \rangle$ **by** auto
hence $\exists v'. (v' \leq move\ ts\ ts''\ v) \wedge (ts'',v' \models re(c) \wedge re(e))$
using view.somewhere-leq
by meson

then obtain v' where v' -def:
 $(v' \leq \text{move } ts \ ts'' \ v) \wedge (ts'', v' \models re(c) \wedge re(e))$
by blast
with backwards-res-act have $ts', v' \models (re(c) \vee cl(c)) \wedge re(e)$
using e-def backwards-res-stab nequals
by (metis (no-types, lifting) d-def eq2)
hence $\exists v'. (v' \leq \text{move } ts \ ts'' \ v) \wedge (ts', v' \models (re(c) \vee cl(c)) \wedge (re(e)))$
using v' -def by blast
hence $ts', \text{move } ts \ ts'' \ v \models \langle (re(c) \vee cl(c)) \wedge (re(e)) \rangle$
using view.somewhere-leq by meson
hence $ts', \text{move } ts \ ts'' \ v \models \langle re(c) \wedge re(e) \rangle \vee \langle cl(c) \wedge re(e) \rangle$
using hmlsl.somewhere-and-or-distr by blast
thus False
proof
assume $assm': ts', \text{move } ts \ ts'' \ v \models \langle re(c) \wedge re(e) \rangle$
have $ts', \text{move } ts \ ts'' \ v \models \neg (c = e)$ using nequals by blast
thus False using $assm'$ cr-res.IH e-def move-stab by fastforce
next
assume $assm': ts', \text{move } ts \ ts'' \ v \models \langle cl(c) \wedge re(e) \rangle$
hence $ts', \text{move } ts \ ts'' \ v \models \neg (c = e) \wedge \langle cl(c) \wedge re(e) \rangle$
using e-def nequals by blast
hence $ts', \text{move } ts \ ts'' \ v \models \neg (c = e) \wedge \langle cl(c) \wedge (re(e) \vee cl(e)) \rangle$
by blast
hence $pcc: ts', \text{move } ts \ ts'' \ v \models pcc \ e \ c$ by blast
have $ts', \text{move } ts \ ts'' \ v \models (\exists d. pcc \ d \ c) \rightarrow \Box r(c) \perp$
using local-LC move-stab by fastforce
hence $ts', \text{move } ts \ ts'' \ v \models \Box r(c) \perp$ using pcc by blast
thus $ts'', \text{move } ts \ ts'' \ v \models \perp$ using e-trans by blast
qed
qed
qed
qed
next
case (cr-clm $ts' \ ts''$)
have $\text{move } ts \ ts' \ v = \text{move } ts' \ ts'' \ (\text{move } ts \ ts' \ v)$
using traffic.move-stability-clm cr-clm.hyps traffic.move-trans
by auto
hence move-stab: $\text{move } ts \ ts' \ v = \text{move } ts \ ts'' \ v$
by (metis traffic.abstract.simps cr-clm.hyps traffic.move-trans)
show ?case
proof (rule)
assume e-def: $(ts'', \text{move } ts \ ts'' \ v \models \langle re(c) \wedge re(e) \rangle)$
obtain d where d-def: $\exists n. (ts' - c(d, n) \rightarrow ts'')$
using cr-clm.hyps by blast
from this obtain n where n-def: $(ts' - c(d, n) \rightarrow ts'')$ by blast
from e-def have $\exists v'. (v' \leq \text{move } ts \ ts'' \ v) \wedge (ts'', v' \models re(c) \wedge re(e))$
using view.somewhere-leq by fastforce
then obtain v' where v' -def: $(v' \leq \text{move } ts \ ts'' \ v) \wedge (ts'', v' \models re(c) \wedge re(e))$
by blast

then have $(ts', v' \models re(c) \wedge re(e))$
using *n-def backwards-c-res-stab* **by** *blast*
then have $ts', move\ ts\ ts''\ v \models \langle re(c) \wedge re(e) \rangle$
using *v'-def view.somewhere-leq* **by** *meson*
thus *False* **using** *cr-clm.IH move-stab e-def nequals* **by** *fastforce*
qed
next
case $(wd-res\ ts'\ ts'')$
have $move\ ts\ ts'\ v = move\ ts'\ ts''\ (move\ ts\ ts'\ v)$
using *traffic.move-stability-wdr wd-res.hyps traffic.move-trans*
by *auto*
hence *move-stab: move ts ts' v = move ts ts'' v*
by $(metis\ traffic.abstract.simps\ wd-res.hyps\ traffic.move-trans)$
show *?case*
proof $(rule)$
assume *e-def: $(ts'', move\ ts\ ts''\ v \models \langle re(c) \wedge re(e) \rangle)$*
obtain *d* **where** *d-def: $\exists n. (ts' -wdr(d, n) \rightarrow ts'')$*
using *wd-res.hyps* **by** *blast*
from *this* **obtain** *n* **where** *n-def: $(ts' -wdr(d, n) \rightarrow ts'')$* **by** *blast*
from *e-def* **have** $\exists v'. (v' \leq move\ ts\ ts''\ v) \wedge (ts'', v' \models re(c) \wedge re(e))$
using *view.somewhere-leq* **by** *fastforce*
then obtain *v'* **where** *v'-def: $(v' \leq move\ ts\ ts''\ v) \wedge (ts'', v' \models re(c) \wedge re(e))$*
by *blast*
then have $(ts', v' \models re(c) \wedge re(e))$
using *n-def backwards-wdr-res-stab* **by** *blast*
then have $(ts', move\ ts\ ts''\ v \models \langle re(c) \wedge re(e) \rangle)$
using *v'-def view.somewhere-leq* **by** *meson*
thus *False* **using** *wd-res.IH move-stab* **by** *fastforce*
qed
next
case $(wd-clm\ ts'\ ts'')$
have $move\ ts\ ts'\ v = move\ ts'\ ts''\ (move\ ts\ ts'\ v)$
using *traffic.move-stability-wdc wd-clm.hyps traffic.move-trans*
by *auto*
hence *move-stab: move ts ts' v = move ts ts'' v*
by $(metis\ traffic.abstract.simps\ wd-clm.hyps\ traffic.move-trans)$
show *?case*
proof $(rule)$
assume *e-def: $(ts'', move\ ts\ ts''\ v \models \langle re(c) \wedge re(e) \rangle)$*
obtain *d* **where** *d-def: $(ts' -wdc(d) \rightarrow ts'')$*
using *wd-clm.hyps* **by** *blast*
from *e-def* **have** $\exists v'. (v' \leq move\ ts\ ts''\ v) \wedge (ts'', v' \models re(c) \wedge re(e))$
using *view.somewhere-leq* **by** *fastforce*
then obtain *v'* **where** *v'-def: $(v' \leq move\ ts\ ts''\ v) \wedge (ts'', v' \models re(c) \wedge re(e))$*
by *blast*
then have $(ts', v' \models re(c) \wedge re(e))$
using *d-def backwards-wdc-res-stab* **by** *blast*
hence $ts', move\ ts\ ts''\ v \models \langle re(c) \wedge re(e) \rangle$
using *v'-def view.somewhere-leq* **by** *meson*

```

    thus False using wd-clm.IH move-stab by fastforce
  qed
qed
qed

```

While the safety theorem was only proven for a single car, we can show that the choice of this car is irrelevant. That is, if we have a safe situation, and switch the perspective to another car, the resulting situation is also safe.

lemma *safety-switch-invariant*: $\models(\forall e. \text{safe}(e)) \rightarrow @c (\forall e. \text{safe}(e))$

proof (*rule allI*|*rule impI*) $+$

fix *ts v v'*

fix *e d :: cars*

assume *assm*: $ts, v \models \forall e. \text{safe}(e)$

and *v'-def*: $(v = c > v')$

and *nequals*: $ts, v \models \neg(d = e)$

show $ts, v' \models \neg \langle re(d) \wedge re(e) \rangle$

proof(*rule*)

assume *e-def*: $ts, v' \models \langle re(d) \wedge re(e) \rangle$

from *e-def* **obtain** *v'sub* **where** *v'sub-def*:

$(v'sub \leq v') \wedge (ts, v'sub \models re(d) \wedge re(e))$

using *view.somewhere-leq* **by** *fastforce*

have *own v' = c* **using** *v'-def view.switch-def* **by** *auto*

hence *own v'sub = c* **using** *v'sub-def less-eq-view-ext-def* **by** *auto*

obtain *vsub* **where** *vsub*: $(vsub = c > v'sub) \wedge (vsub \leq v)$

using *v'-def v'sub-def view.switch-leq* **by** *blast*

from *v'sub-def* **and** *vsub* **have** $ts, vsub \models @c re(d)$

by (*metis view.switch-unique*)

hence *vsub-re-d*: $ts, vsub \models re(d)$ **using** *at-res-inst* **by** *blast*

from *v'sub-def* **and** *vsub* **have** $ts, vsub \models @c re(e)$

by (*metis view.switch-unique*)

hence *vsub-re-e*: $ts, vsub \models re(e)$ **using** *at-res-inst* **by** *blast*

hence $ts, vsub \models re(d) \wedge re(e)$ **using** *vsub-re-e vsub-re-d* **by** *blast*

hence $ts, v \models \langle re(d) \wedge re(e) \rangle$

using *vsub view.somewhere-leq* **by** *fastforce*

then show *False* **using** *assm nequals* **by** *blast*

qed

qed

end

end

14 Regular Sensors

This section contains an instantiations of the sensor function for "regular sensors". That is, each car can perceive its own physical size and braking distance. However, it can only perceive the physical size of other cars, and does not know about their braking distance.

theory *Regular-Sensors*

```

imports ../Length
begin

definition regular::cars  $\Rightarrow$  traffic  $\Rightarrow$  cars  $\Rightarrow$  real
  where regular e ts c  $\equiv$ 
    if (e = c) then traffic.physical-size ts c + traffic.braking-distance ts c
    else traffic.physical-size ts c

locale regular-sensors = traffic + view
begin

interpretation regular-sensors: sensors regular :: cars  $\Rightarrow$  traffic  $\Rightarrow$  cars  $\Rightarrow$  real
proof unfold-locales
  fix e ts c
  show 0 < regular e ts c
  by (metis (no-types, opaque-lifting) less-add-same-cancel2 less-trans regular-def
    traffic.psGeZero traffic.sdGeZero)
qed

notation regular-sensors.space (space)
notation regular-sensors.len (len)

Similar to the situation with perfect sensors, we can show that the perceived
length of a car is independent of the spatial transitions between traffic snap-
shots. The length may only change during evolutions, in particular if the
car changes its dynamical behaviour.

lemma create-reservation-length-stable:
  (ts-r(d) $\rightarrow$ ts')  $\longrightarrow$  len v ts c = len v ts' c
proof
  assume assm:(ts-r(d) $\rightarrow$ ts')
  hence eq:space ts v c = space ts' v c
  using traffic.create-reservation-def sensors.space-def regular-def
  by (simp add: regular-sensors.sensors-axioms)
  show len v ( ts ) c = len v ( ts' ) c
  proof (cases left ((space ts v) c) > right (ext v))
    assume outside-right:left ((space ts v) c) > right (ext v)
    hence outside-right':left ((space ts' v) c) > right (ext v) using eq by simp
    from outside-right and outside-right' show ?thesis
    by (simp add: regular-sensors.len-def eq)
  next
  assume inside-right: $\neg$  left ((space ts v) c) > right (ext v)
  hence inside-right': $\neg$  left ((space ts' v) c) > right (ext v) using eq by simp
  show len v ( ts ) c = len v ( ts' ) c
  proof (cases left (ext v) > right ((space ts v) c))
    assume outside-left:left (ext v) > right ((space ts v) c)
    hence outside-left':left (ext v) > right ((space ts' v) c) using eq by simp
    from outside-left and outside-left' show ?thesis
    by (simp add: regular-sensors.len-def eq)
  next

```


assume *inside-left*: \neg left (ext v) > right ((space ts v) c)
hence *inside-left'*: \neg left (ext v) > right ((space ts' v) c) **using** eq **by** simp
from *inside-left inside-right inside-left' inside-right'* eq
show ?thesis **by** (simp add: regular-sensors.len-def)
qed
qed
qed

lemma *create-claim-length-stable*:

$(ts - c(d, n) \rightarrow ts') \longrightarrow \text{len } v \text{ } ts \text{ } c = \text{len } v \text{ } ts' \text{ } c$

proof

assume *assm*: $(ts - c(d, n) \rightarrow ts')$

hence eq:space ts v c = space ts' v c

using traffic.create-claim-def sensors.space-def regular-def

by (simp add: regular-sensors.sensors-axioms)

show len v (ts) c = len v (ts') c

proof (cases left ((space ts v) c) > right (ext v))

assume *outside-right*:left ((space ts v) c) > right (ext v)

hence *outside-right'*:left ((space ts' v) c) > right (ext v) **using** eq **by** simp

from *outside-right* **and** *outside-right'* **show** ?thesis

by (simp add: regular-sensors.len-def eq)

next

assume *inside-right*: \neg left ((space ts v) c) > right (ext v)

hence *inside-right'*: \neg left ((space ts' v) c) > right (ext v) **using** eq **by** simp

show len v (ts) c = len v (ts') c

proof (cases left (ext v) > right ((space ts v) c))

assume *outside-left*: left (ext v) > right ((space ts v) c)

hence *outside-left'*: left (ext v) > right ((space ts' v) c) **using** eq **by** simp

from *outside-left* **and** *outside-left'* **show** ?thesis

by (simp add: regular-sensors.len-def eq)

next

assume *inside-left*: \neg left (ext v) > right ((space ts v) c)

hence *inside-left'*: \neg left (ext v) > right ((space ts' v) c) **using** eq **by** simp

from *inside-left inside-right inside-left' inside-right'* eq

show ?thesis **by** (simp add: regular-sensors.len-def)

qed

qed

qed

lemma *withdraw-reservation-length-stable*:

$(ts - wdr(d, n) \rightarrow ts') \longrightarrow \text{len } v \text{ } ts \text{ } c = \text{len } v \text{ } ts' \text{ } c$

proof

assume *assm*: $(ts - wdr(d, n) \rightarrow ts')$

hence eq:space ts v c = space ts' v c

using traffic.withdraw-reservation-def sensors.space-def regular-def

by (simp add: regular-sensors.sensors-axioms)

show len v (ts) c = len v (ts') c

proof (cases left ((space ts v) c) > right (ext v))

assume *outside-right*:left ((space ts v) c) > right (ext v)

hence *outside-right'*: $\text{left}((\text{space } ts' v) c) > \text{right}(ext v)$ **using** *eq* **by** *simp*
from *outside-right* **and** *outside-right'* **show** *?thesis*
 by (*simp add: regular-sensors.len-def eq*)

next
 assume *inside-right*: $\neg \text{left}((\text{space } ts v) c) > \text{right}(ext v)$
 hence *inside-right'*: $\neg \text{left}((\text{space } ts' v) c) > \text{right}(ext v)$ **using** *eq* **by** *simp*
show $\text{len } v (ts) c = \text{len } v (ts') c$
proof (*cases left (ext v) > right ((space ts v) c)*)
 assume *outside-left*: $\text{left}(ext v) > \text{right}((\text{space } ts v) c)$
 hence *outside-left'*: $\text{left}(ext v) > \text{right}((\text{space } ts' v) c)$ **using** *eq* **by** *simp*
from *outside-left* **and** *outside-left'* **show** *?thesis*
 by (*simp add: regular-sensors.len-def eq*)

next
 assume *inside-left*: $\neg \text{left}(ext v) > \text{right}((\text{space } ts v) c)$
 hence *inside-left'*: $\neg \text{left}(ext v) > \text{right}((\text{space } ts' v) c)$ **using** *eq* **by** *simp*
from *inside-left* *inside-right* *inside-left'* *inside-right'* *eq*
show *?thesis* **by** (*simp add: regular-sensors.len-def*)

qed
qed
qed

lemma *withdraw-claim-length-stable*:
 $(ts - wdc(d) \rightarrow ts') \rightarrow \text{len } v ts c = \text{len } v ts' c$

proof
 assume *assm*: $(ts - wdc(d) \rightarrow ts')$
 hence *eq*: $\text{space } ts v c = \text{space } ts' v c$
using *traffic.withdraw-claim-def sensors.space-def regular-def*
 by (*simp add: regular-sensors.sensors-axioms*)
show $\text{len } v (ts) c = \text{len } v (ts') c$
proof (*cases left ((space ts v) c) > right (ext v)*)
 assume *outside-right*: $\text{left}((\text{space } ts v) c) > \text{right}(ext v)$
 hence *outside-right'*: $\text{left}((\text{space } ts' v) c) > \text{right}(ext v)$ **using** *eq* **by** *simp*
from *outside-right* **and** *outside-right'* **show** *?thesis*
 by (*simp add: regular-sensors.len-def eq*)

next
 assume *inside-right*: $\neg \text{left}((\text{space } ts v) c) > \text{right}(ext v)$
 hence *inside-right'*: $\neg \text{left}((\text{space } ts' v) c) > \text{right}(ext v)$ **using** *eq* **by** *simp*
show $\text{len } v (ts) c = \text{len } v (ts') c$
proof (*cases left (ext v) > right ((space ts v) c)*)
 assume *outside-left*: $\text{left}(ext v) > \text{right}((\text{space } ts v) c)$
 hence *outside-left'*: $\text{left}(ext v) > \text{right}((\text{space } ts' v) c)$ **using** *eq* **by** *simp*
from *outside-left* **and** *outside-left'* **show** *?thesis*
 by (*simp add: regular-sensors.len-def eq*)

next
 assume *inside-left*: $\neg \text{left}(ext v) > \text{right}((\text{space } ts v) c)$
 hence *inside-left'*: $\neg \text{left}(ext v) > \text{right}((\text{space } ts' v) c)$ **using** *eq* **by** *simp*
from *inside-left* *inside-right* *inside-left'* *inside-right'* *eq*
show *?thesis* **by** (*simp add: regular-sensors.len-def*)

qed

qed
qed

Since the perceived length of cars depends on the owner of the view, we can now prove how this perception changes if we change the perspective of a view.

lemma *sensors-le*: $e \neq c \longrightarrow \text{regular } e \text{ ts } c < \text{regular } c \text{ ts } c$
using *traffic.sdGeZero* **by** (*simp add: regular-def*)

lemma *sensors-leq*: $\text{regular } e \text{ ts } c \leq \text{regular } c \text{ ts } c$
by (*metis less-eq-real-def regular-sensors.sensors-le*)

lemma *space-eq*: $\text{own } v = \text{own } v' \longrightarrow \text{space ts } v \text{ } c = \text{space ts } v' \text{ } c$
using *regular-sensors.space-def sensors-def* **by** *auto*

lemma *switch-space-le*: $(\text{own } v) \neq c \wedge (v=c>v') \longrightarrow \text{space ts } v \text{ } c < \text{space ts } v' \text{ } c$
proof

assume *assm*: $(\text{own } v) \neq c \wedge (v=c>v')$

hence *sens:regular* $(\text{own } v) \text{ ts } c < \text{regular } (\text{own } v') \text{ ts } c$

using *sensors-le view.switch-def* **by** *auto*

then have *le:pos* $\text{ts } c + \text{regular } (\text{own } v) \text{ ts } c < \text{pos ts } c + \text{regular } (\text{own } v') \text{ ts } c$
by *auto*

have *left-eq:left* $(\text{space ts } v \text{ } c) = \text{left } (\text{space ts } v' \text{ } c)$

using *regular-sensors.left-space* **by** *auto*

have *r1:right* $(\text{space ts } v \text{ } c) = \text{pos ts } c + \text{regular } (\text{own } v) \text{ ts } c$

using *regular-sensors.right-space* **by** *auto*

have *r2:right* $(\text{space ts } v' \text{ } c) = \text{pos ts } c + \text{regular } (\text{own } v') \text{ ts } c$

using *regular-sensors.right-space* **by** *auto*

then have *right* $(\text{space ts } v \text{ } c) < \text{right } (\text{space ts } v' \text{ } c)$

using *r1 r2 le* **by** *auto*

then have *left* $(\text{space ts } v' \text{ } c) \geq \text{left } (\text{space ts } v \text{ } c)$

$\wedge (\text{right } (\text{space ts } v \text{ } c) \leq \text{right } (\text{space ts } v' \text{ } c))$

$\wedge \neg(\text{left } (\text{space ts } v \text{ } c) \geq \text{left } (\text{space ts } v' \text{ } c))$

$\wedge \text{right } (\text{space ts } v' \text{ } c) \leq \text{right } (\text{space ts } v \text{ } c)$

using *regular-sensors.left-space left-eq* **by** *auto*

then show $\text{space ts } v \text{ } c < \text{space ts } v' \text{ } c$

using *less-real-int-def left-eq* **by** *auto*

qed

lemma *switch-space-leq*: $(v=c>v') \longrightarrow \text{space ts } v \text{ } c \leq \text{space ts } v' \text{ } c$
by (*metis less-imp-le order-refl switch-space-le view.switch-refl view.switch-unique*)

end

end

15 HMLSL for Regular Sensors

Within this section, we instantiate HMLSL for cars with regular sensors.

theory *HMLSL-Regular*

```

imports ../HMLSL Regular-Sensors
begin

locale hmlsl-regular = regular-sensors + restriction
begin
interpretation hmlsl : hmlsl regular :: cars  $\Rightarrow$  traffic  $\Rightarrow$  cars  $\Rightarrow$  real
proof unfold-locales
  fix e ts c
  show  $0 < \text{regular } e \text{ } ts \text{ } c$ 
    by (metis less-add-same-cancel2 less-trans regular-def
      traffic.psGeZero traffic.sdGeZero)
qed

notation hmlsl.re (re'(-'))
notation hmlsl.cl (cl'(-'))
notation hmlsl.len (len)

```

The spatial atoms are dependent of the perspective of the view, hence we cannot prove similar lemmas as for perfect sensors.

However, we can still prove lemmas corresponding to the activity and stability rules of the proof system for MSL [5].

Similar to the situation with perfect sensors, needed to instantiate the sensor function, to ensure that the perceived length does not change during spatial transitions.

lemma *backwards-res-act*:

$$(ts - r(c) \rightarrow ts') \wedge (ts', v \models re(c)) \longrightarrow (ts, v \models re(c) \vee cl(c))$$

proof

assume *assm*: $(ts - r(c) \rightarrow ts') \wedge (ts', v \models re(c))$

from *assm* **have** *len-eq*: $len \ v \ ts \ c = len \ v \ ts' \ c$

using *create-reservation-length-stable* **by** *blast*

have *res* $ts \ c \sqsubseteq res \ ts' \ c$ **using** *assm traffic.create-res-subseteq1*

by *auto*

hence *restr-sub-res*: $restrict \ v \ (res \ ts) \ c \sqsubseteq restrict \ v \ (res \ ts') \ c$

using *assm restriction.restrict-view* **by** *auto*

have *clm* $ts \ c \sqsubseteq res \ ts' \ c$ **using** *assm traffic.create-res-subseteq2*

using *assm restriction.restrict-view* **by** *auto*

hence *restr-sub-clm*: $restrict \ v \ (clm \ ts) \ c \sqsubseteq restrict \ v \ (res \ ts') \ c$

using *assm restriction.restrict-view* **by** *auto*

have $restrict \ v \ (res \ ts) \ c = \emptyset \vee restrict \ v \ (res \ ts) \ c \neq \emptyset$ **by** *simp*

then show $ts, v \models (re(c) \vee cl(c))$

proof

assume *restr-res-nonempty*: $restrict \ v \ (res \ ts) \ c \neq \emptyset$

hence *restrict-one*: $|restrict \ v \ (res \ ts) \ c| = 1$

using *nat-int.card-non-empty-geq-one nat-int.card-subset-le dual-order.antisym*
restr-sub-res assm **by** *fastforce*

have $restrict \ v \ (res \ ts) \ c \sqsubseteq lan \ v$ **using** *restr-sub-res assm* **by** *auto*

hence $restrict \ v \ (res \ ts) \ c = lan \ v$ **using** *restriction.restrict-eq-lan-sub*
restrict-one assm **by** *auto*

```

then show ?thesis using assm len-eq by auto
next
assume restr-res-empty: restrict v (res ts) c = ∅
then have clm-non-empty: restrict v (clm ts) c ≠ ∅
  by (metis assm bot.extremum inf.absorb1 inf-commute local.hmlsl.free-no-clm
    restriction.create-reservation-restrict-union restriction.restrict-def
    un-empty-absorb1)
then have restrict-one: |restrict v (clm ts) c| = 1
using nat-int.card-non-empty-geq-one nat-int.card-subset-le dual-order.antisym
  restr-subs-clm assm by fastforce
have restrict v (clm ts) c ⊆ lan v using restr-subs-clm assm by auto
hence restrict v (clm ts) c = lan v using restriction.restrict-eq-lan-subs
  restrict-one assm by auto
then show ?thesis using assm len-eq by auto
qed
qed

```

lemma *backwards-res-act-somewhere*:
 $(ts - r(c) \rightarrow ts') \wedge (ts', v \models \langle re(c) \rangle) \longrightarrow (ts, v \models \langle re(c) \vee cl(c) \rangle)$
using *backwards-res-act* **by** *blast*

lemma *backwards-res-stab*:
 $(ts - r(d) \rightarrow ts') \wedge (d \neq c) \wedge (ts', v \models re(c)) \longrightarrow (ts, v \models re(c))$
using *regular-sensors.create-reservation-length-stable restrict-def'*
traffic.create-res-subseteq1-neq **by** *auto*

lemma *backwards-c-res-stab*:
 $(ts - c(d, n) \rightarrow ts') \wedge (ts', v \models re(c)) \longrightarrow (ts, v \models re(c))$
using *create-claim-length-stable traffic.create-clm-eq-res*
by (*metis (mono-tags, lifting) traffic.create-claim-def*)

lemma *backwards-wdc-res-stab*:
 $(ts - wdc(d) \rightarrow ts') \wedge (ts', v \models re(c)) \longrightarrow (ts, v \models re(c))$
using *withdraw-claim-length-stable traffic.withdraw-clm-eq-res*
by (*metis (mono-tags, lifting) traffic.withdraw-claim-def*)

lemma *backwards-wdr-res-stab*:
 $(ts - wdr(d, n) \rightarrow ts') \wedge (ts', v \models re(c)) \longrightarrow (ts, v \models re(c))$
by (*metis inf.absorb1 order-trans regular-sensors.withdraw-reservation-length-stable*
restrict-def' restriction.restrict-res traffic.withdraw-res-subseteq)

We now proceed to prove the *reservation lemma*, which was crucial in the manual safety proof [2].

lemma *reservation1*: $\models (re(c) \vee cl(c)) \rightarrow \Box r(c) re(c)$
proof (*rule allI* | *rule impI*) +
fix *ts v ts'*
assume *assm: ts, v \models re(c) \vee cl(c)* **and** *ts'-def: ts - r(c) \rightarrow ts'*
from *assm* **show** *ts', v \models re(c)*

```

proof
  assume  $re:ts,v \models re(c)$ 
  then show ?thesis
  by (metis inf.absorb1 order-trans regular-sensors.create-reservation-length-stable
       restrict-def' restriction.restrict-subseteq traffic.create-res-subseteq1 ts'-def)
next
  assume  $cl:ts,v \models cl(c)$ 
  then show ?thesis
  by (metis inf.absorb1 order-trans regular-sensors.create-reservation-length-stable
       restrict-def' restriction.restrict-subseteq traffic.create-res-subseteq2 ts'-def)
qed
qed

lemma reservation2:  $\models(\Box r(c) \ re(c)) \rightarrow (re(c) \vee cl(c))$ 
  using backwards-res-act traffic.always-create-res
  by metis

lemma reservation:  $\models(\Box r(c) \ re(c)) \leftrightarrow (re(c) \vee cl(c))$ 
  using reservation1 reservation2 by blast
end
end

```

16 Safety for Cars with Regular Sensors

This section contains the definition of requirements for lane change and distance controllers for cars, with the assumption of regular sensors. Using these definitions, we show that safety is an invariant along all possible behaviour of cars. However, we need to slightly amend our notion of safety, compared to the safety proof for perfect sensors.

```

theory Safety-Regular
  imports HMLSL-Regular
begin
context hmlsl-regular
begin

interpretation hmlsl : hmlsl regular :: cars  $\Rightarrow$  traffic  $\Rightarrow$  cars  $\Rightarrow$  real
proof unfold-locales
  fix  $e \ ts \ c$ 
  show  $0 < regular \ e \ ts \ c$ 
  by (metis less-add-same-cancel2 less-trans regular-def
       traffic.psGeZero traffic.sdGeZero)
qed
notation hmlsl.space (space)
notation hmlsl.re (re'(-))
notation hmlsl.cl (cl'(-))
notation hmlsl.len (len)

```

First we show that the same "safety" theorem as for perfect sensors can be

proven. However, we will subsequently show that this theorem does not ensure safety from the perspective of each car.

The controller definitions for this "flawed" safety are the same as for perfect sensors.

abbreviation $safe::cars \Rightarrow \sigma$
where $safe\ e \equiv \forall\ c.\ \neg(c = e) \rightarrow \neg\langle re(c) \wedge re(e) \rangle$

abbreviation $DC::\sigma$
where $DC \equiv \mathbf{G}(\forall\ c\ d.\ \neg(c = d) \rightarrow \neg\langle re(c) \wedge re(d) \rangle) \rightarrow \Box\tau\ \neg\langle re(c) \wedge re(d) \rangle$

abbreviation $pcc::cars \Rightarrow cars \Rightarrow \sigma$
where $pcc\ c\ d \equiv \neg(c = d) \wedge \langle cl(d) \wedge (re(c) \vee cl(c)) \rangle$

abbreviation $LC::\sigma$
where $LC \equiv \mathbf{G}(\forall\ d.(\exists\ c.\ pcc\ c\ d) \rightarrow \Box r(d) \perp)$

The safety proof is exactly the same as for perfect sensors. Note in particular, that we fix a single car e for which we show safety.

theorem $safety\text{-}flawed: \models (\forall\ e.\ safe\ e) \wedge DC \wedge LC \rightarrow \mathbf{G}(\forall\ e.\ safe\ e)$

proof (*rule allI|rule impI*)⁺

fix $ts\ v\ ts'$

fix $e\ c::\ cars$

assume $assm:ts,v \models (\forall\ e.\ safe\ e) \wedge DC \wedge LC$

assume $abs:(ts \Rightarrow ts')$

assume $neg:ts,v \models \neg(c = e)$

from $assm$ **have** $init:ts,v \models (\forall\ e.\ safe\ e)$ **by** *simp*

from $assm$ **have** $DC :ts,v \models DC$ **by** *simp*

from $assm$ **have** $LC:ts,v \models LC$ **by** *simp*

show $ts',move\ ts\ ts'\ v \models \neg\langle re(c) \wedge re(e) \rangle$ **using** abs

proof (*induction*)

case (*refl*)

have $move\ ts\ ts'\ v = v$ **using** *move-nothing* **by** *simp*

thus *?case* **using** *init move-nothing neg* **by** *simp*

next

case (*evolve* $ts'\ ts''$)

have *local-DC*:

$ts',move\ ts\ ts'\ v \models \forall\ c\ d.\ \neg(c = d) \rightarrow \neg\langle re(c) \wedge re(d) \rangle \rightarrow \Box\tau\ \neg\langle re(c) \wedge re(d) \rangle$

using *evolve.hyps DC* **by** *simp*

show *?case*

proof (*rule*)

assume $e\text{-def}: (ts'',move\ ts\ ts''\ v \models \langle re(c) \wedge re(e) \rangle)$

from *evolve.IH* **and** $e\text{-def}$ **and** neg **have**

$ts'\text{-safe}:ts',move\ ts\ ts'\ v \models \neg(c = e) \wedge \neg\langle re(c) \wedge re(e) \rangle$ **by** *blast*

hence *no-coll-after-evol*: $ts',move\ ts\ ts'\ v \models \Box\tau\ \neg\langle re(c) \wedge re(e) \rangle$

using *local-DC* **by** *blast*

have $move\text{-eq}:move\ ts'\ ts''\ (move\ ts\ ts'\ v) = move\ ts\ ts''\ v$

using *evolve.hyps abstract.evolve abstract.refl move-trans*
by (*meson traffic.abstract.evolve traffic.abstract.refl traffic.move-trans*)
from *no-coll-after-evol* **and** *evolve.hyps* **have**
 $ts'', \text{move } ts' \ ts'' \ (\text{move } ts \ ts' \ v) \models \neg \langle re(c) \wedge re(e) \rangle$
by *blast*
thus *False* **using** *e-def* **using** *move-eq* **by** *fastforce*
qed
next
case (*cr-res ts' ts''*)
have *local-LC*: $ts', \text{move } ts \ ts' \ v \models (\forall d. (\exists c. \text{pcc } c \ d) \rightarrow \Box r(d) \perp)$
using *LC cr-res.hyps* **by** *blast*
have $\text{move } ts \ ts' \ v = \text{move } ts' \ ts'' \ (\text{move } ts \ ts' \ v)$
using *move-stability-res cr-res.hyps move-trans*
by *auto*
hence *move-stab*: $\text{move } ts \ ts' \ v = \text{move } ts \ ts'' \ v$
using *cr-res.hyps local.create-reservation-def local.move-def* **by** *auto*
show *?case*
proof (*rule*)
assume *e-def*: $(ts'', \text{move } ts \ ts'' \ v \models \langle re(c) \wedge re(e) \rangle)$
obtain *d* **where** *d-def*: $ts' -r(d) \rightarrow ts''$ **using** *cr-res.hyps* **by** *blast*
have $d = e \vee d \neq e$ **by** *simp*
thus *False*
proof
assume *eq*: $d = e$
hence *e-trans*: $ts' -r(e) \rightarrow ts''$ **using** *d-def* **by** *simp*
from *e-def* **have** $ts'', \text{move } ts \ ts'' \ v \models \langle re(c) \wedge re(e) \rangle$ **by** *auto*
hence $\exists v'. (v' \leq \text{move } ts \ ts'' \ v) \wedge (ts'', v' \models re(c) \wedge re(e))$
using *somewhere-leq*
by *meson*
then obtain *v'* **where** *v'-def*:
 $(v' \leq \text{move } ts \ ts'' \ v) \wedge (ts'', v' \models re(c) \wedge re(e))$
by *blast*
with *backwards-res-act* **have** $ts', v' \models re(c) \wedge (re(e) \vee cl(e))$
using *e-def backwards-res-stab neg*
by (*metis (no-types, lifting) d-def eq*)
hence $\exists v'. (v' \leq \text{move } ts \ ts'' \ v) \wedge (ts', v' \models re(c) \wedge (re(e) \vee cl(e)))$
using *v'-def* **by** *blast*
hence $ts', \text{move } ts \ ts'' \ v \models \langle re(c) \wedge (re(e) \vee cl(e)) \rangle$
using *somewhere-leq* **by** *meson*
hence $ts', \text{move } ts \ ts'' \ v \models \langle re(c) \wedge re(e) \rangle \vee \langle re(c) \wedge cl(e) \rangle$
using *hmlsl.somewhere-and-or-distr* **by** *blast*
thus *False*
proof
assume *assm'*: $ts', \text{move } ts \ ts'' \ v \models \langle re(c) \wedge re(e) \rangle$
have $ts', \text{move } ts \ ts' \ v \models \neg (c = e)$ **using** *neg* **by** *blast*
thus *False* **using** *assm' cr-res.IH e-def move-stab* **by** *force*
next
assume *assm'*: $ts', \text{move } ts \ ts'' \ v \models \langle re(c) \wedge cl(e) \rangle$
hence $ts', \text{move } ts \ ts'' \ v \models \neg (c = e) \wedge \langle re(c) \wedge cl(e) \rangle$

using *neg by force*
 hence $ts', move\ ts\ ts''\ v \models \neg (c = e) \wedge \langle cl(e) \wedge (re(c) \vee cl(c)) \rangle$ **by blast**
 hence $pcc:ts', move\ ts\ ts''\ v \models pcc\ c\ e$ **by blast**
 have $ts', move\ ts\ ts''\ v \models (\exists\ c. pcc\ c\ e) \rightarrow \Box r(e) \perp$
 using *local-LC move-stab by fastforce*
 hence $ts', move\ ts\ ts''\ v \models \Box r(e) \perp$ **using pcc by blast**
 thus $ts'', move\ ts\ ts''\ v \models \perp$ **using e-trans by blast**
qed
next
 assume *neg:d ≠ e*
 have $c=d \vee c \neq d$ **by simp**
 thus *False*
proof
 assume *neg2:c ≠ d*
 from *e-def* have $ts'', move\ ts\ ts''\ v \models \langle re(c) \wedge re(e) \rangle$ **by auto**
 hence $\exists v'. (v' \leq move\ ts\ ts''\ v) \wedge (ts'', v' \models re(c) \wedge re(e))$
 using *somewhere-leq*
 by meson
 then obtain *v' where v'-def:*
 $(v' \leq move\ ts\ ts''\ v) \wedge (ts'', v' \models re(c) \wedge re(e))$
 by blast
 with *backwards-res-stab* have *overlap:* $ts', v' \models re(c) \wedge (re(e))$
 using *e-def backwards-res-stab neg neg2*
 by (metis (no-types, lifting) d-def neg)
 hence *unsafe2:* $ts', move\ ts\ ts''\ v \models \langle re(c) \wedge re(e) \rangle$
 using *somewhere-leq v'-def by blast*
 from *cr-res.IH* have $ts', move\ ts\ ts''\ v \models \neg \langle re(c) \wedge re(e) \rangle$
 using *move-stab by force*
 thus *False using unsafe2 by best*
next
 assume *eq2:c = d*
 hence *e-trans:* $ts' -r(c) \rightarrow ts''$ **using d-def by simp**
 from *e-def* have $ts'', move\ ts\ ts''\ v \models \langle re(c) \wedge re(e) \rangle$ **by auto**
 hence $\exists v'. (v' \leq move\ ts\ ts''\ v) \wedge (ts'', v' \models re(c) \wedge re(e))$
 using *somewhere-leq*
 by meson
 then obtain *v' where v'-def:*
 $(v' \leq move\ ts\ ts''\ v) \wedge (ts'', v' \models re(c) \wedge re(e))$
 by blast
 with *backwards-res-act* have $ts', v' \models (re(c) \vee cl(c)) \wedge (re(e))$
 using *e-def backwards-res-stab neg*
 by (metis (no-types, lifting) d-def eq2)
 hence $\exists v'. (v' \leq move\ ts\ ts''\ v) \wedge (ts', v' \models (re(c) \vee cl(c)) \wedge (re(e)))$
 using *v'-def by blast*
 hence $ts', move\ ts\ ts''\ v \models \langle (re(c) \vee cl(c)) \wedge (re(e)) \rangle$
 using *somewhere-leq by meson*
 hence $ts', move\ ts\ ts''\ v \models \langle re(c) \wedge re(e) \rangle \vee \langle cl(c) \wedge re(e) \rangle$
 using *hmlsl.somewhere-and-or-distr by blast*
 thus *False*

```

proof
  assume  $assm':ts',move\ ts\ ts''\ v \models \langle re(c) \wedge re(e) \rangle$ 
  have  $ts',move\ ts\ ts''\ v \models \neg (c = e)$  using neg by blast
  thus False using  $assm'\ cr-res.IH\ e-def\ move-stab$  by fastforce
next
  assume  $assm':ts',move\ ts\ ts''\ v \models \langle cl(c) \wedge re(e) \rangle$ 
  hence  $ts',move\ ts\ ts''\ v \models \neg (c = e) \wedge \langle cl(c) \wedge re(e) \rangle$ 
    using neg by blast
  hence  $ts',move\ ts\ ts''\ v \models \neg (c = e) \wedge \langle cl(c) \wedge (re(e) \vee cl(e)) \rangle$ 
    by blast
  hence  $pcc:ts',move\ ts\ ts''\ v \models pcc\ e\ c$  by blast
  have  $ts',move\ ts\ ts''\ v \models (\exists d. pcc\ d\ c) \rightarrow \Box r(c) \perp$ 
    using local-LC\ move-stab by fastforce
  hence  $ts',move\ ts\ ts''\ v \models \Box r(c) \perp$  using pcc by blast
  thus  $ts'',move\ ts\ ts''\ v \models \perp$  using e-trans by blast
  qed
qed
qed
qed
next
  case (cr-clm  $ts'\ ts''$ )
  have  $move\ ts\ ts'\ v = move\ ts'\ ts''\ (move\ ts\ ts'\ v)$ 
    using move-stability-clm\ cr-clm.hyps\ move-trans
    by auto
  hence move-stab:  $move\ ts\ ts'\ v = move\ ts\ ts''\ v$ 
    by (metis\ abstract.simps\ cr-clm.hyps\ move-trans)
  show ?case
  proof (rule)
    assume e-def:  $(ts'',move\ ts\ ts''\ v \models \langle re(c) \wedge re(e) \rangle)$ 
    obtain  $d$  where d-def:  $\exists n. (ts' - c(d,n) \rightarrow ts'')$  using cr-clm.hyps by blast
    then obtain  $n$  where n-def:  $(ts' - c(d,n) \rightarrow ts'')$  by blast
    from e-def have  $\exists v'. (v' \leq move\ ts\ ts''\ v) \wedge (ts'',v' \models re(c) \wedge re(e))$ 
      using somewhere-leq by fastforce
    then obtain  $v'$  where v'-def:
       $(v' \leq move\ ts\ ts''\ v) \wedge (ts'',v' \models re(c) \wedge re(e))$ 
      by blast
    then have  $(ts',v' \models re(c) \wedge re(e))$ 
      using n-def\ backwards-c-res-stab by blast
    hence  $ts',move\ ts\ ts''\ v \models \langle re(c) \wedge re(e) \rangle$ 
      using e-def\ v'-def\ somewhere-leq by meson
    thus False using cr-clm.IH\ move-stab by fastforce
  qed
next
  case (wd-res  $ts'\ ts''$ )
  have  $move\ ts\ ts'\ v = move\ ts'\ ts''\ (move\ ts\ ts'\ v)$ 
    using move-stability-wdr\ wd-res.hyps\ move-trans
    by auto
  hence move-stab:  $move\ ts\ ts'\ v = move\ ts\ ts''\ v$ 
    by (metis\ abstract.simps\ wd-res.hyps\ move-trans)

```

```

show ?case
proof (rule)
  assume e-def:  $(ts'', \text{move } ts \ ts'' \ v \models \langle re(c) \wedge re(e) \rangle)$ 
  obtain d and n where n-def:  $(ts' - \text{wdr}(d, n) \rightarrow ts'')$ 
    using wd-res.hypos by auto
  from e-def have  $\exists v'. (v' \leq \text{move } ts \ ts'' \ v) \wedge (ts'', v' \models re(c) \wedge re(e))$ 
    using somewhere-leq by fastforce
  then obtain v' where v'-def:
     $(v' \leq \text{move } ts \ ts'' \ v) \wedge (ts'', v' \models re(c) \wedge re(e))$ 
    by blast
  then have  $(ts', v' \models re(c) \wedge re(e))$ 
    using n-def backwards-wdr-res-stab by blast
  hence  $ts', \text{move } ts \ ts'' \ v \models \langle re(c) \wedge re(e) \rangle$  using
    v'-def somewhere-leq by meson
  thus False using wd-res.IH move-stab by fastforce
qed
next
case (wd-clm ts' ts'')
have  $\text{move } ts \ ts' \ v = \text{move } ts' \ ts'' \ (\text{move } ts \ ts' \ v)$ 
  using move-stability-wdc wd-clm.hypos move-trans
  by auto
hence move-stab:  $\text{move } ts \ ts' \ v = \text{move } ts \ ts'' \ v$ 
  by (metis abstract.simps wd-clm.hypos move-trans)
show ?case
proof (rule)
  assume e-def:  $(ts'', \text{move } ts \ ts'' \ v \models \langle re(c) \wedge re(e) \rangle)$ 
  obtain d where d-def:  $(ts' - \text{wdc}(d) \rightarrow ts'')$  using wd-clm.hypos by blast
  from e-def have  $\exists v'. (v' \leq \text{move } ts \ ts'' \ v) \wedge (ts'', v' \models re(c) \wedge re(e))$ 
    using somewhere-leq by fastforce
  then obtain v' where v'-def:
     $(v' \leq \text{move } ts \ ts'' \ v) \wedge (ts'', v' \models re(c) \wedge re(e))$ 
    by blast
  from this have  $(ts', v' \models re(c) \wedge re(e))$ 
    using d-def backwards-wdc-res-stab by blast
  hence  $ts', \text{move } ts \ ts'' \ v \models \langle re(c) \wedge re(e) \rangle$  using v'-def somewhere-leq by
meson
  thus False using wd-clm.IH move-stab by fastforce
qed
qed
qed

```

As stated above, the flawed safety theorem does not ensure safety for the perspective of each car. In particular, we can construct a traffic snapshot and a view, such that it satisfies our safety predicate for each car, but if we switch the perspective of the view to another car, the situation is unsafe. A visualisation of this situation can be found in the publication of this work at iFM 2017 [4].

lemma *safety-not-invariant-switch*:

$$\exists ts \ v. (ts, v \models \forall e. \text{safe}(e) \wedge (\exists c. @c \neg(\forall e. \text{safe}(e))))$$

proof –

```

obtain  $d\ c :: \text{cars}$  where  $\text{assumption: } d \neq c$ 
  using  $\text{cars.at-least-two-cars-exists}$  by  $\text{best}$ 
obtain  $\text{pos}'$  where  $\text{pos}'\text{-def: } \forall (c :: \text{cars}). (\text{pos}'\ c) = (5 :: \text{real})$  by  $\text{best}$ 
obtain  $\text{po}$  where  $\text{pos}'\text{-def: } \text{po} = (\text{pos}'(c:=0))(d:=2)$  by  $\text{best}$ 
obtain  $\text{re}$  where  $\text{res-def: } \forall (c :: \text{cars}). \text{re } c = \text{Abs-nat-int}\{0\}$  by  $\text{best}$ 
obtain  $\text{cl}$  where  $\text{clm-def: } \forall (c :: \text{cars}). \text{cl } c = \emptyset$  by  $\text{best}$ 
obtain  $\text{dy}$  where  $\text{dyn-def: } \forall c :: \text{cars}. \forall x :: \text{real}. (\text{dy } c) x = (0 :: \text{real})$  by  $\text{force}$ 
obtain  $\text{sd}$  where  $\text{sd-def: } \forall (c :: \text{cars}). \text{sd } c = (2 :: \text{real})$  by  $\text{best}$ 
obtain  $\text{ps}$  where  $\text{ps-def: } \forall (c :: \text{cars}). \text{ps } c = (1 :: \text{real})$  by  $\text{best}$ 
obtain  $\text{ts-rep}$  where  $\text{ts-rep-def: } \text{ts-rep} = (\text{po}, \text{re}, \text{cl}, \text{dy}, \text{ps}, \text{sd})$  by  $\text{best}$ 

have  $\text{disj: } \forall c. ((\text{re } c) \sqcap (\text{cl } c) = \emptyset)$  by  $(\text{simp add: clm-def nat-int.inter-empty1})$ 
have  $\text{re-geq-one: } \forall c. |\text{re } c| \geq 1$ 
  by  $(\text{simp add: Abs-nat-int-inverse nat-int.card'-def res-def})$ 
have  $\text{re-leq-two: } \forall c. |\text{re } c| \leq 2$ 
  using  $\text{nat-int.card'.rep-eq res-def nat-int.rep-single}$  by  $\text{auto}$ 
have  $\text{cl-leq-one: } \forall c. |\text{cl } c| \leq 1$ 
  using  $\text{nat-int.card-empty-zero clm-def}$  by  $(\text{simp})$ 
have  $\text{add-leq-two: } \forall c. |\text{re } c| + |\text{cl } c| \leq 2$ 
  using  $\text{nat-int.card-empty-zero clm-def re-leq-two}$  by  $(\text{simp})$ 
have  $\text{clNextRe} :$ 
   $\forall c. ((\text{cl } c) \neq \emptyset \longrightarrow (\exists n. \text{Rep-nat-int } (\text{re } c) \cup \text{Rep-nat-int } (\text{cl } c) = \{n, n+1\}))$ 
  by  $(\text{simp add: clm-def})$ 
from  $\text{dyn-def}$  have  $\text{dyn-geq-zero: } \forall c. \forall x. (\text{dy } c) x \geq 0$  by  $\text{auto}$ 
from  $\text{ps-def}$  have  $\text{ps-ge-zero: } \forall c. \text{ps } c > 0$  by  $\text{auto}$ 
from  $\text{sd-def}$  have  $\text{sd-ge-zero: } \forall c. \text{sd } c > 0$  by  $\text{auto}$ 
have  $\text{ts-in-type: } \text{ts-rep} \in$ 
 $\{ \text{ts} :: (\text{cars} \Rightarrow \text{real}) * (\text{cars} \Rightarrow \text{lanes}) * (\text{cars} \Rightarrow \text{lanes}) * (\text{cars} \Rightarrow \text{real} \Rightarrow \text{real}) * (\text{cars} \Rightarrow \text{real}) * (\text{cars} \Rightarrow \text{real}).$ 
   $(\forall c. ((\text{fst } (\text{snd } \text{ts})))\ c \sqcap ((\text{fst } (\text{snd } (\text{snd } \text{ts}))))\ c = \emptyset) \wedge$ 
   $(\forall c. |(\text{fst } (\text{snd } \text{ts}))\ c| \geq 1) \wedge$ 
   $(\forall c. |(\text{fst } (\text{snd } \text{ts}))\ c| \leq 2) \wedge$ 
   $(\forall c. |(\text{fst } (\text{snd } (\text{snd } \text{ts}))\ c)| \leq 1) \wedge$ 
   $(\forall c. |(\text{fst } (\text{snd } \text{ts}))\ c| + |(\text{fst } (\text{snd } (\text{snd } \text{ts})))\ c| \leq 2) \wedge$ 
   $(\forall c. (\text{fst } (\text{snd } (\text{snd } (\text{snd } (\text{snd } (\text{snd } \text{ts}))))))\ c \neq \emptyset \longrightarrow$ 
   $(\exists n. \text{Rep-nat-int } ((\text{fst } (\text{snd } \text{ts}))\ c) \cup \text{Rep-nat-int } ((\text{fst } (\text{snd } (\text{snd } \text{ts})))\ c)$ 
   $= \{n, n+1\}) \wedge$ 
   $(\forall c. \text{fst } (\text{snd } (\text{snd } (\text{snd } (\text{snd } (\text{snd } \text{ts}))))))\ c > 0) \wedge$ 
   $(\forall c. \text{snd } (\text{snd } (\text{snd } (\text{snd } (\text{snd } \text{ts}))))))\ c > 0$ 
 $\}$ 
  using  $\text{pos-def res-def clm-def disj re-geq-one re-leq-two cl-leq-one add-leq-two}$ 
   $\text{ps-ge-zero sd-ge-zero ts-rep-def}$ 
  by  $\text{auto}$ 
obtain  $v$  where  $\text{v-def:}$ 
   $\text{v} = (\text{ext} = \text{Abs-real-int } (0, 3), \text{lan} = \text{Abs-nat-int}\{0\}, \text{own} = d)$ 
  by  $\text{best}$ 
obtain  $\text{ts}$  where  $\text{ts-def: } \text{ts} = \text{Abs-traffic } \text{ts-rep}$ 
  by  $\text{blast}$ 
have  $\text{size: } \forall c. \text{physical-size } \text{ts } c = 1$  using  $\text{ps-def physical-size-def ts-rep-def}$ 

```

ts-in-type ts-def ps-ge-zero **using** *Abs-traffic-inverse*
by auto

have safe: $ts, v \models \forall e. \text{safe}(e)$

proof

have other-len-zero: $\forall e. e \neq c \wedge e \neq d \longrightarrow \| \text{len } v \text{ ts } e \| = 0$

proof (*rule allI*|*rule impI*)**+**

fix e

assume $e\text{-def}: e \neq c \wedge e \neq d$

have position: $\text{pos } ts \ e = 5$ **using** $e\text{-def}$ $ts\text{-def}$ $ts\text{-rep-def}$ $ts\text{-in-type}$ $ts\text{-def}$
Abs-traffic-inverse pos-def fun-upd-apply $\text{pos}'\text{-def}$ traffic.pos-def

by auto

have regular (*own* v) $ts \ e = 1$

using $e\text{-def}$ $v\text{-def}$ sensors-def $ps\text{-def}$ $ts\text{-def}$ size regular-def **by auto**

then have space: $\text{space } ts \ v \ e = \text{Abs-real-int } (5, 6)$

using $e\text{-def}$ pos-def $\text{position hmlsl.space-def}$ **by auto**

have left ($\text{space } ts \ v \ e$) $>$ $\text{right } (\text{ext } v)$

using $\text{space } v\text{-def}$ $\text{Abs-real-int-inverse}$ **by auto**

thus $\| \text{len } v \text{ ts } e \| = 0$

using hmlsl.len-def $\text{real-int.length-def}$ $\text{Abs-real-int-inverse}$ **by auto**

qed

have no-cars: $\forall e. e \neq c \wedge e \neq d \longrightarrow (ts, v \models \neg \langle \text{re}(e) \vee \text{cl}(e) \rangle)$

proof (*rule allI*|*rule impI*|*rule notI*)**+**

fix e

assume $\text{neq}: e \neq c \wedge e \neq d$

assume $\text{contra}: ts, v \models \langle \text{re}(e) \vee \text{cl}(e) \rangle$

from other-len-zero **have** $\text{len-e}: \| \text{len } v \text{ ts } e \| = 0$ **using** neq **by auto**

from contra **obtain** v' **where** $v'\text{-def}: v' \leq v \wedge (ts, v' \models \text{re}(e) \vee \text{cl}(e))$

using somewhere-leq **by force**

from $v'\text{-def}$ **and** len-e **have** $\text{len-v}': \| \text{len } v' \text{ ts } e \| = 0$

using $\text{hmlsl.len-empty-subview}$ **by blast**

from $v'\text{-def}$ **have** $ts, v' \models \text{re}(e) \vee \text{cl}(e)$ **by blast**

thus False **using** $\text{len-v}'$ **by auto**

qed

have sensors-c: $\text{regular } (\text{own } v) \text{ ts } c = 1$

using $v\text{-def}$ regular-def $ps\text{-def}$ $ts\text{-def}$ size assumption **by auto**

have space-c: $\text{space } ts \ v \ c = \text{Abs-real-int } (0, 1)$

using pos-def $ts\text{-def}$ $ts\text{-rep-def}$ $ts\text{-in-type}$ $\text{Abs-traffic-inverse}$

fun-upd-apply sensors-c $\text{assumption hmlsl.space-def}$ traffic.pos-def

by auto

have lc: $\text{left } (\text{space } ts \ v \ c) = 0$ **using** space-c $\text{Abs-real-int-inverse}$ **by auto**

have rv: $\text{right } (\text{ext } v) = 3$ **using** $v\text{-def}$ $\text{Abs-real-int-inverse}$ **by auto**

have lv: $\text{left } (\text{ext } v) = 0$ **using** $v\text{-def}$ $\text{Abs-real-int-inverse}$ **by auto**

have rc: $\text{right } (\text{space } ts \ v \ c) = 1$ **using** space-c $\text{Abs-real-int-inverse}$ **by auto**

have len-c: $\text{len } v \text{ ts } c = \text{Abs-real-int}(0, 1)$

using space-c $v\text{-def}$ hmlsl.len-def lc lv rv rc **by auto**

have sensors-d: $\text{regular } (\text{own } v) \text{ ts } d = 3$

using $v\text{-def}$ regular-def $\text{braking-distance-def}$ $ts\text{-def}$ size sd-def

```

    Abs-traffic-inverse ts-in-type ts-rep-def
  by auto
have space-d:space ts v d = Abs-real-int(2,5)
  using pos-def ts-def ts-rep-def ts-in-type Abs-traffic-inverse
    fun-upd-apply sensors-d assumption hmlsl.space-def traffic.pos-def
  by auto
have ld:left (space ts v d) = 2 using space-d Abs-real-int-inverse by auto
have rd:right (space ts v d) = 5 using space-d Abs-real-int-inverse by auto
have len-d:len v ts d = Abs-real-int(2,3)
  using space-d v-def hmlsl.len-def ld rd lv rv by auto
have no-overlap-c-d:ts,v ⊨¬ ⟨re(c) ∧ re(d)⟩
proof (rule notI)
  assume contra:ts,v ⊨ ⟨re(c) ∧ re(d)⟩
  obtain v' where v'-def:(v' ≤ v) ∧ (ts,v' ⊨re(c) ∧ re(d))
    using somewhere-leq contra by force
  hence len-eq:len v' ts c = len v' ts d by simp
  from v'-def have v'-c:||len v' ts c|| > 0
    and v'-d:||len v' ts d|| > 0 by simp+
  from v'-c have v'-rel-c:
    left (space ts v' c) < right (ext v') ∧ right (space ts v' c) > left (ext v')
    using hmlsl.len-non-empty-inside by blast
  from v'-d have v'-rel-d:
    left (space ts v' d) < right (ext v') ∧ right (space ts v' d) > left (ext v')
    using hmlsl.len-non-empty-inside by blast
  have less-len:len v' ts c ≤ len v ts c
    using hmlsl.view-leq-len-leq v'-c v'-def less-eq-view-ext-def by blast
  have sp-eq-c:space ts v' c = space ts v c
    using v'-def less-eq-view-ext-def regular-def hmlsl.space-def by auto
  have sp-eq-d:space ts v' d = space ts v d
    using v'-def less-eq-view-ext-def regular-def hmlsl.space-def by auto

  have right (ext v') > 0 ∧ right (ext v') > 2
    using ld lc v'-rel-c v'-rel-d sp-eq-c sp-eq-d by auto
  hence r-v':right (ext v') > 2 by blast
  have left (ext v') < 1 ∧ left (ext v') < 5
    using rd rc v'-rel-c v'-rel-d sp-eq-c sp-eq-d by auto
  hence l-v':left (ext v') < 1 by blast
  have len v' ts c ≠ ext v'
proof
  assume len v' ts c = ext v'
  hence eq:right (len v' ts c) = right (ext v') by simp
  from less-len have right (len v' ts c) ≤ right (len v ts c)
    by (simp add: less-eq-real-int-def)
  with len-c have right (len v' ts c) ≤ 1
    using Abs-real-int-inverse by auto
  thus False using r-v' eq by linarith
qed
thus False using v'-def by blast
qed

```

```

fix x
show  $ts, v \models \text{safe}(x)$ 
proof (rule allI|rule impI)+
  fix y
  assume  $x\text{-neg-}y: ts, v \models \neg(y = x)$ 
  show  $ts, v \models \neg \langle \text{re}(y) \wedge \text{re}(x) \rangle$ 
  proof (cases  $y \neq c \wedge y \neq d$ )
    assume  $y \neq c \wedge y \neq d$ 
    hence  $(ts, v \models \neg \langle \text{re}(y) \vee \text{cl}(y) \rangle)$  using no-cars by blast
    hence  $ts, v \models \neg \langle \text{re}(y) \rangle$  by blast
    then show ?thesis by blast
  next
  assume  $\neg(y \neq c \wedge y \neq d)$ 
  hence  $y = c \vee y = d$  by blast
  thus ?thesis
  proof
    assume  $y\text{-eq-}c: y = c$ 
    thus ?thesis
    proof (cases  $x = d$ )
      assume  $x = d$ 
      then show  $ts, v \models \neg \langle \text{re}(y) \wedge \text{re}(x) \rangle$ 
        using no-overlap-c-d y-eq-c by blast
    next
      assume  $x : x \neq d$ 
      have  $x2: x \neq c$  using y-eq-c x-neg-y by blast
      hence  $(ts, v \models \neg \langle \text{re}(x) \vee \text{cl}(x) \rangle)$  using no-cars x by blast
      hence  $ts, v \models \neg \langle \text{re}(x) \rangle$  by blast
      thus ?thesis by blast
    qed
  next
  assume  $y\text{-eq-}c: y = d$ 
  thus ?thesis
  proof (cases  $x = c$ )
    assume  $x = c$ 
    thus  $ts, v \models \neg \langle \text{re}(y) \wedge \text{re}(x) \rangle$  using no-overlap-c-d y-eq-c by blast
  next
    assume  $x : x \neq c$ 
    have  $x2: x \neq d$  using y-eq-c x-neg-y by blast
    hence  $(ts, v \models \neg \langle \text{re}(x) \vee \text{cl}(x) \rangle)$  using no-cars x by blast
    hence  $ts, v \models \neg \langle \text{re}(x) \rangle$  by blast
    thus ?thesis by blast
  qed
qed
qed
qed
qed
qed

```

have $\text{unsafe}: ts, v \models (\exists c. (@c \neg (\forall e. \text{safe}(e))))$
proof –

```

have  $ts, v \models (@c \neg(\forall e. safe(e)))$ 
proof (rule allI|rule impI|rule notI)+
  fix  $vc$ 
  assume  $sw:(v=c > vc)$ 
  have  $spatial\text{-}vc: ext\ v = ext\ vc \wedge lan\ v = lan\ vc$ 
    using  $switch\text{-}def\ sw$  by  $blast$ 
  assume  $safe: ts, vc \models (\forall e. safe(e))$ 
  obtain  $vc'$  where  $vc'\text{-}def:$ 
     $vc' = (ext = Abs\text{-}real\text{-}int\ (2,3), lan = Abs\text{-}nat\text{-}int\ \{0\}, own = c)$ 
    by  $best$ 
  have  $own\text{-}eq: own\ vc' = own\ vc$  using  $sw\ switch\text{-}def\ vc'\text{-}def$  by  $auto$ 
  have  $ext\text{-}vc: ext\ vc = Abs\text{-}real\text{-}int\ (0,3)$  using  $spatial\text{-}vc\ v\text{-}def$  by  $force$ 
  have  $right\text{-}ok: right\ (ext\ vc) \geq right\ (ext\ vc')$ 
    using  $vc'\text{-}def\ ext\text{-}vc\ Abs\text{-}real\text{-}int\text{-}inverse$  by  $auto$ 
  have  $left\text{-}ok: left\ (ext\ vc') \geq left\ (ext\ vc)$ 
    using  $vc'\text{-}def\ ext\text{-}vc\ Abs\text{-}real\text{-}int\text{-}inverse$  by  $auto$ 
  hence  $ext\text{-}leq: ext\ vc' \leq ext\ vc$ 
    using  $right\text{-}ok\ left\text{-}ok\ less\text{-}eq\text{-}real\text{-}int\text{-}def$  by  $auto$ 
  have  $lan\ vc = Abs\text{-}nat\text{-}int\ \{0\}$  using  $v\text{-}def\ switch\text{-}def\ sw$  by  $force$ 
  hence  $lan\text{-}leq: lan\ vc' \sqsubseteq lan\ vc$  using  $vc'\text{-}def\ order\text{-}refl$  by  $force$ 
  have  $leqvc: vc' \leq vc$ 
    using  $ext\text{-}leq\ lan\text{-}leq\ own\text{-}eq\ less\text{-}eq\text{-}view\text{-}ext\text{-}def$  by  $force$ 
  have  $sensors\text{-}c: regular\ (own\ vc')\ ts\ c = 3$ 
    using  $vc'\text{-}def\ regular\text{-}def\ ps\text{-}def\ traffic.braking\text{-}distance\text{-}def$ 
     $ts\text{-}def\ sd\text{-}def\ size\ assumption\ Abs\text{-}traffic\text{-}inverse\ ts\text{-}in\text{-}type\ ts\text{-}rep\text{-}def$ 
    by  $auto$ 
  have  $space\text{-}c: space\ ts\ vc'\ c = Abs\text{-}real\text{-}int\ (0,3)$ 
    using  $pos\text{-}def\ ts\text{-}def\ ts\text{-}rep\text{-}def\ ts\text{-}in\text{-}type\ Abs\text{-}traffic\text{-}inverse$ 
     $fun\text{-}upd\text{-}apply\ sensors\text{-}c\ assumption\ hmlsl.space\text{-}def\ traffic.pos\text{-}def$ 
    by  $auto$ 
  have  $lc: left\ (space\ ts\ vc'\ c) = 0$  using  $space\text{-}c\ Abs\text{-}real\text{-}int\text{-}inverse$  by  $auto$ 
  have  $rv: right\ (ext\ vc') = 3$  using  $vc'\text{-}def\ Abs\text{-}real\text{-}int\text{-}inverse$  by  $auto$ 
  have  $lv: left\ (ext\ vc') = 2$  using  $vc'\text{-}def\ Abs\text{-}real\text{-}int\text{-}inverse$  by  $auto$ 
  have  $rc: right\ (space\ ts\ vc'\ c) = 3$  using  $space\text{-}c\ Abs\text{-}real\text{-}int\text{-}inverse$  by  $auto$ 
  have  $len\text{-}c: len\ vc'\ ts\ c = Abs\text{-}real\text{-}int\ (2,3)$ 
    using  $space\text{-}c\ v\text{-}def\ hmlsl.len\text{-}def\ lc\ lv\ rv\ rc$  by  $auto$ 
  have  $res\text{-}c: restrict\ vc'\ (res\ ts)\ c = Abs\text{-}nat\text{-}int\ \{0\}$ 
    using  $ts\text{-}def\ ts\text{-}rep\text{-}def\ ts\text{-}in\text{-}type\ Abs\text{-}traffic\text{-}inverse\ res\text{-}def\ traffic.res\text{-}def$ 
     $inf\text{-}idem\ restrict\text{-}def\ vc'\text{-}def$ 
    by  $force$ 
  have  $sensors\text{-}d: regular\ (own\ vc')\ ts\ d = 1$ 
    using  $vc'\text{-}def\ regular\text{-}def\ ts\text{-}def\ size\ sd\text{-}def\ Abs\text{-}traffic\text{-}inverse\ ts\text{-}in\text{-}type$ 
     $ts\text{-}rep\text{-}def\ assumption$ 
    by  $auto$ 
  have  $space\text{-}d: space\ ts\ vc'\ d = Abs\text{-}real\text{-}int\ (2,3)$ 
    using  $pos\text{-}def\ ts\text{-}def\ ts\text{-}rep\text{-}def\ ts\text{-}in\text{-}type\ Abs\text{-}traffic\text{-}inverse$ 
     $fun\text{-}upd\text{-}apply\ sensors\text{-}d\ assumption\ hmlsl.space\text{-}def\ traffic.pos\text{-}def$ 
    by  $auto$ 
  have  $ld: left\ (space\ ts\ vc'\ d) = 2$  using  $space\text{-}d\ Abs\text{-}real\text{-}int\text{-}inverse$  by  $auto$ 

```



```

have rd: right (space ts vc' d) = 3 using space-d Abs-real-int-inverse by auto
have len-d :len vc' ts d = Abs-real-int(2,3)
  using space-d v-def hmlsl.len-def ld rd lv rv
  by auto
have res-d:restrict vc' (res ts) d = Abs-nat-int {0}
  using ts-def ts-rep-def ts-in-type Abs-traffic-inverse res-def traffic.res-def
  inf-idem restrict-def vc'-def by force
have ts,vc' ⊨ re(c) ∧ re(d) using
  len-d len-c vc'-def ts-def ts-rep-def ts-in-type Abs-traffic-inverse
  res-c res-d nat-int.card'-def
  Abs-real-int-inverse real-int.length-def traffic.res-def
  nat-int.singleton2 Abs-nat-int-inverse
  by auto
with leqvc have ts,vc ⊨ ⟨re(c) ∧ re(d)⟩ using somewhere-leq by blast
with assumption have ts,vc ⊨ ¬(c = d) ∧ ⟨re(c) ∧ re(d)⟩ by blast
with safe show False by blast
qed
thus ?thesis by blast
qed
from safe and unsafe have ts,v ⊨ ∀ e. safe(e) ∧ (∃ c. (@c ¬(∀ e. safe(e))))
  by blast
thus ?thesis by blast
qed

```

Now we show how to amend the controller specifications to gain safety as an invariant even with regular sensors.

The distance controller can be strengthened, by requiring that we switch to the perspective of one of the cars involved first, before checking for the collision. Since all variables are universally quantified, this ensures that no collision exists for the perspective of any car.

abbreviation $DC'::\sigma$

where $DC' \equiv \mathbf{G} (\forall c d. \neg(c = d) \rightarrow$
 $(@d \neg\langle re(c) \wedge re(d) \rangle) \rightarrow \Box \tau @d \neg\langle re(c) \wedge re(d) \rangle)$

The amendment to the lane change controller is slightly different. Instead of checking the potential collision only from the perspective of the car d trying to change lanes, we require that also no other car may perceive a potential collision. Note that the restriction to d 's behaviour can only be enforced within d , if the information from the other car is somehow passed to d . Hence, we require the cars to communicate in some way. However, we do not need to specify, *how* this communication is implemented.

abbreviation $LC'::\sigma$

where $LC' \equiv \mathbf{G} (\forall d. (\exists c. (@c (pcc c d)) \vee (@d (pcc c d))) \rightarrow \Box r(d) \perp)$

With these new controllers, we can prove a stronger theorem than before. Instead of proving safety from the perspective of a single car as previously, we now only consider a traffic situation to be safe, if it satisfies the safety

predicate from the perspective of *all* cars. Note that this immediately implies the safety invariance theorem proven for perfect sensors.

theorem *safety*: $\models (\forall e. @e (safe\ e)) \wedge DC' \wedge LC' \rightarrow \mathbf{G}(\forall e. @e (safe\ e))$

proof (*rule allI; rule allI; rule impI; rule allI; rule impI; rule allI*)

fix *ts v ts' e*

assume *assm*: $ts, v \models (\forall e. @e (safe\ e)) \wedge DC' \wedge LC'$

assume *abs*: $(ts \Rightarrow ts')$

from *assm* **have** *init*: $ts, v \models (\forall e. @e (safe\ e))$ **by** *simp*

from *assm* **have** *DC*: $ts, v \models DC'$ **by** *simp*

from *assm* **have** *LC*: $ts, v \models LC'$ **by** *simp*

show *ts', move ts ts' v* $\models (@e (safe\ e))$ **using** *abs*

proof (*induction*)

case (*refl*)

have *move ts ts v = v* **using** *move-nothing* **by** *blast*

thus *?case* **using** *move-nothing init* **by** *simp*

next

case (*evolve ts' ts''*)

have *local-DC*:

$ts', move\ ts\ ts'\ v \models \forall\ c\ d. \neg(c = d) \rightarrow$
 $(@d \neg\langle re(c) \wedge re(d) \rangle) \rightarrow (\Box\tau\ @d \neg\langle re(c) \wedge re(d) \rangle)$

using *evolve.hyps DC* **by** *simp*

show *?case*

proof (*rule ccontr*)

assume $\neg (ts'', move\ ts\ ts''\ v \models (@e (safe\ e)))$

then have *e-def*: $ts'', move\ ts\ ts''\ v \models \neg(@e (safe\ e))$ **by** *best*

hence *ts'', move ts ts'' v* $\models @e (\neg\ safe\ e)$

using *switch-always-exists switch-unique* **by** *fastforce*

then obtain *ve* **where** *ve-def*:

$((move\ ts\ ts''\ v) = e > ve) \wedge (ts'', ve \models \neg\ safe\ e)$

using *switch-always-exists* **by** *fastforce*

hence *unsafe*: $ts'', ve \models \exists\ c. \neg(c = e) \wedge \langle re(c) \wedge re(e) \rangle$ **by** *blast*

then obtain *c* **where** *c-def*: $ts'', ve \models \neg(c = e) \wedge \langle re(c) \wedge re(e) \rangle$ **by** *blast*

from *evolve.IH* **and** *c-def* **have**

ts'-safe: $ts', move\ ts\ ts'\ v \models @e (\neg(c = e) \wedge \neg\langle re(c) \wedge re(e) \rangle)$

by *blast*

hence *not-eq*: $ts', move\ ts\ ts'\ v \models @e (\neg(c = e))$

and *safe'*: $ts', move\ ts\ ts'\ v \models @e (\neg\langle re(c) \wedge re(e) \rangle)$

using *hmlsl.at-conj-distr* **by** *simp+*

from *not-eq* **have** *not-eq-v*: $ts', move\ ts\ ts'\ v \models \neg(c = e)$

using *hmlsl.at-eq switch-always-exists* **by** *auto*

have

$ts', move\ ts\ ts'\ v \models \neg(c = e) \rightarrow$
 $(@e \neg\langle re(c) \wedge re(e) \rangle) \rightarrow (\Box\tau\ @e \neg\langle re(c) \wedge re(e) \rangle)$

using *local-DC* **by** *simp*

hence *dc*: $ts', move\ ts\ ts'\ v \models (@e \neg\langle re(c) \wedge re(e) \rangle) \rightarrow$

$(\Box\tau\ @e \neg\langle re(c) \wedge re(e) \rangle)$

using *not-eq-v*

by *simp*

hence *no-coll-after-evol*: $ts', move\ ts\ ts'\ v \models (\Box\tau\ @e \neg\langle re(c) \wedge re(e) \rangle)$

```

    using safe'
    by simp
  hence 1:ts'',move ts' ts'' (move ts ts' v)  $\models$  @e  $\neg$   $\langle$ re(c)  $\wedge$  re(e) $\rangle$ 
    using evolve.hyps by simp
  have move-eq:move ts' ts'' (move ts ts' v) = move ts ts'' v
    using evolve.hyps abstract.evolve abstract.refl move-trans
    by blast
  from 1 have ts'', move ts ts'' v  $\models$  @e  $\neg$   $\langle$ re(c)  $\wedge$  re(e) $\rangle$ 
    using move-eq by fastforce
  hence ts'',ve  $\models$   $\neg$   $\langle$ re(c)  $\wedge$  re(e) $\rangle$  using ve-def by blast
  thus False using c-def by blast
qed
next
case (cr-clm ts' ts'')
have move ts ts' v = move ts' ts'' (move ts ts' v)
  using move-stability-clm cr-clm.hyps move-trans
  by auto
hence move-stab: move ts ts' v = move ts ts'' v
  by (metis abstract.simps cr-clm.hyps move-trans)
show ?case
proof (rule ccontr)
  assume  $\neg$  (ts'',move ts ts'' v  $\models$  ( @e (safe e)))
  then have e-def:ts'',move ts ts'' v  $\models$   $\neg$ (@e (safe e)) by best
  hence ts'',move ts ts'' v  $\models$  @e ( $\neg$  safe e)
    using switch-always-exists switch-unique
    by fastforce
  then obtain ve where ve-def:
    ((move ts ts'' v) =e> ve)  $\wedge$  (ts'',ve  $\models$   $\neg$  safe e)
    using switch-always-exists by fastforce
  hence unsafe:ts'',ve  $\models$   $\exists$  c.  $\neg$ (c = e)  $\wedge$   $\langle$ re(c)  $\wedge$  re(e) $\rangle$  by blast
  then obtain c where c-def:ts'',ve  $\models$   $\neg$ (c = e)  $\wedge$   $\langle$ re(c)  $\wedge$  re(e) $\rangle$ 
    by blast
  hence c-neq-e:ts'',ve  $\models$   $\neg$  (c = e) by blast
  obtain d n where d-def: (ts' -c(d,n)  $\rightarrow$  ts'') using cr-clm.hyps by blast
  from c-def have  $\exists$  v'. (v'  $\leq$  ve)  $\wedge$  (ts'',v'  $\models$  re(c)  $\wedge$  re(e))
    using somewhere-leq by fastforce
  then obtain v' where v'-def:(v'  $\leq$  ve)  $\wedge$  (ts'',v'  $\models$  re(c)  $\wedge$  re(e))
    by blast
  then have (ts',v'  $\models$  re(c)  $\wedge$  re(e))
    using d-def backwards-c-res-stab by blast
  hence ts',ve  $\models$   $\neg$  safe (e)
    using c-neq-e c-def v'-def somewhere-leq by meson
  thus False using cr-clm.IH move-stab ve-def by fastforce
qed
next
case (wd-res ts' ts'')
have move ts ts' v = move ts' ts'' (move ts ts' v)
  using move-stability-wdr wd-res.hyps move-trans
  by auto

```

hence *move-stab*: $move\ ts\ ts'\ v = move\ ts\ ts''\ v$
by (*metis abstract.simps wd-res.hyps move-trans*)
show ?*case*
proof (*rule ccontr*)
assume $\neg (ts'', move\ ts\ ts''\ v \models (@e\ (safe\ e)))$
then have $e-def: ts'', move\ ts\ ts''\ v \models \neg (@e\ (safe\ e))$ **by** *best*
hence $ts'', move\ ts\ ts''\ v \models @e\ (\neg\ safe\ e)$
using *switch-always-exists switch-unique* **by** (*fastforce*)
then obtain *ve* **where** *ve-def*:
 $((move\ ts\ ts''\ v) =e>\ ve) \wedge (ts'', ve \models \neg\ safe\ e)$
using *switch-always-exists* **by** *fastforce*
hence $unsafe: ts'', ve \models \exists\ c.\ \neg(c = e) \wedge \langle re(c) \wedge re(e) \rangle$ **by** *blast*
then obtain *c* **where** $c-def: ts'', ve \models \neg(c = e) \wedge \langle re(c) \wedge re(e) \rangle$ **by** *blast*
hence $c-neq-e: ts'', ve \models \neg(c = e)$ **by** *blast*
obtain *d n* **where** *n-def*:
 $(ts' -wdr(d, n) \rightarrow ts'')$ **using** *wd-res.hyps* **by** *blast*
from *c-def* **have** $\exists v'. (v' \leq ve) \wedge (ts'', v' \models re(c) \wedge re(e))$
using *somewhere-leq* **by** *fastforce*
then obtain *v'* **where** $v'-def: (v' \leq ve) \wedge (ts'', v' \models re(c) \wedge re(e))$
by *blast*
then have $(ts', v' \models re(c) \wedge re(e))$
using *n-def backwards-wdr-res-stab* **by** *blast*
hence $ts', ve \models \neg\ safe\ (e)$
using *c-neq-e c-def v'-def somewhere-leq* **by** *meson*
thus *False* **using** *wd-res.IH move-stab ve-def* **by** *fastforce*
qed
next
case (*wd-clm ts' ts''*)
have $move\ ts\ ts'\ v = move\ ts'\ ts''\ (move\ ts\ ts'\ v)$
using *move-stability-wdc wd-clm.hyps move-trans*
by *auto*
hence *move-stab*: $move\ ts\ ts'\ v = move\ ts\ ts''\ v$
by (*metis abstract.simps wd-clm.hyps move-trans*)
show ?*case*
proof (*rule ccontr*)
assume $\neg (ts'', move\ ts\ ts''\ v \models (@e\ (safe\ e)))$
then have $e-def: ts'', move\ ts\ ts''\ v \models \neg (@e\ (safe\ e))$ **by** *best*
then obtain *ve* **where** *ve-def*:
 $((move\ ts\ ts''\ v) =e>\ ve) \wedge (ts'', ve \models \neg\ safe\ e)$
using *switch-always-exists* **by** *fastforce*
hence $unsafe: ts'', ve \models \exists\ c.\ \neg(c = e) \wedge \langle re(c) \wedge re(e) \rangle$ **by** *blast*
then obtain *c* **where** $c-def: ts'', ve \models \neg(c = e) \wedge \langle re(c) \wedge re(e) \rangle$ **by** *blast*
hence $c-neq-e: ts'', ve \models \neg(c = e)$ **by** *blast*
obtain *d* **where** $d-def: (ts' -wdc(d) \rightarrow ts'')$ **using** *wd-clm.hyps* **by** *blast*
from *c-def* **have** $\exists v'. (v' \leq ve) \wedge (ts'', v' \models re(c) \wedge re(e))$
using *somewhere-leq* **by** *fastforce*
then obtain *v'* **where** *v'-def*:
 $(v' \leq ve) \wedge (ts'', v' \models re(c) \wedge re(e))$ **by** *blast*
then have $(ts', v' \models re(c) \wedge re(e))$

using *d-def backwards-wdc-res-stab* **by** *blast*
hence $ts',ve \models \neg \text{safe } (e)$
using *c-neq-e c-def v'-def somewhere-leq* **by** *meson*
thus *False* **using** *wd-clm.IH move-stab ve-def* **by** *fastforce*
qed
next
case (*cr-res ts' ts''*)
have *local-LC*:
 $ts',move \text{ ts } ts' v \models \forall d. (\exists c. (@c (pcc \ c \ d)) \vee (@d (pcc \ c \ d))) \rightarrow \Box r(d) \perp$
using *LC cr-res.hyps(1)* **by** *blast*
have $move \text{ ts } ts' v = move \text{ ts' } ts'' (move \text{ ts } ts' v)$
using *move-stability-res cr-res.hyps move-trans*
by *auto*
hence *move-stab*: $move \text{ ts } ts' v = move \text{ ts } ts'' v$
by (*metis abstract.simps cr-res.hyps move-trans*)
show *?case*
proof (*rule ccontr*)
obtain *d* **where** *d-def*: $(ts' -r(d) \rightarrow ts'')$
using *cr-res.hyps* **by** *blast*
assume $\neg (ts'',move \text{ ts } ts'' v \models (@e (\text{safe } e)))$
then have *e-def*: $ts'',move \text{ ts } ts'' v \models \neg (@e (\text{safe } e))$ **by** *best*
hence $ts'',move \text{ ts } ts'' v \models @e (\neg \text{safe } e)$
using *switch-always-exists switch-unique* **by** *fast*
then obtain *ve* **where** *ve-def*:
 $((move \text{ ts } ts'' v) =e> ve) \wedge (ts'',ve \models \neg \text{safe } e)$
using *switch-always-exists* **by** *fastforce*
hence *unsafe*: $ts'',ve \models \exists c. \neg(c = e) \wedge \langle re(c) \wedge re(e) \rangle$ **by** *blast*
then obtain *c* **where** *c-def*: $ts'',ve \models \neg(c = e) \wedge \langle re(c) \wedge re(e) \rangle$ **by** *blast*
hence *c-neq-e*: $ts'',ve \models \neg(c = e)$ **by** *blast*
show *False*
proof (*cases d=e*)
case *True*
hence *e-trans*: $ts' -r(e) \rightarrow ts''$ **using** *d-def* **by** *simp*
from *c-def* **have** $ts'',ve \models \langle re(c) \wedge re(e) \rangle$ **by** *auto*
hence $\exists v'. (v' \leq ve) \wedge (ts'',v' \models re(c) \wedge re(e))$
using *somewhere-leq*
by *meson*
then obtain *v'* **where** *v'-def*:
 $(v' \leq ve) \wedge (ts'',v' \models re(c) \wedge re(e))$ **by** *blast*
with *backwards-res-act* **have** $ts',v' \models re(c) \wedge (re(e) \vee cl(e))$
using *c-def backwards-res-stab c-neq-e*
by (*metis (no-types, lifting) d-def True*)
hence $\exists v'. (v' \leq ve) \wedge (ts',v' \models re(c) \wedge (re(e) \vee cl(e)))$
using *v'-def* **by** *blast*
hence $ts',ve \models \langle re(c) \wedge (re(e) \vee cl(e)) \rangle$
using *somewhere-leq* **by** *meson*
hence $ts',ve \models \langle re(c) \wedge re(e) \rangle \vee \langle re(c) \wedge cl(e) \rangle$
using *hmlsl.somewhere-and-or-distr* **by** *metis*
then show *False*

```

proof
  assume  $assm':ts',ve \models \langle re(c) \wedge re(e) \rangle$ 
  have  $ts',move\ ts\ ts'\ v \models \neg (c = e)$  using  $c-def$  by  $blast$ 
  thus  $False$  using  $assm'\ cr-res.IH\ c-def\ move-stab\ ve-def$  by  $force$ 
next
  assume  $assm':ts',ve \models \langle re(c) \wedge cl(e) \rangle$ 
  hence  $ts',ve \models \neg (c = e) \wedge \langle re(c) \wedge cl(e) \rangle$  using  $c-def$  by  $force$ 
  hence  $ts',ve \models pcc\ c\ e$  by  $blast$ 
  hence  $ts',move\ ts\ ts'\ v \models @e\ (pcc\ c\ e)$ 
    using  $ve-def\ move-stab\ switch-unique$  by  $fastforce$ 
  hence  $pcc:ts',move\ ts\ ts'\ v \models (@c\ (pcc\ c\ e)) \vee (@e\ (pcc\ c\ e))$ 
    by  $blast$ 
  have
     $ts',move\ ts\ ts'\ v \models (\exists\ c.(@c\ (pcc\ c\ e)) \vee (@e\ (pcc\ c\ e))) \rightarrow \Box r(e) \perp$ 
    using  $local-LC\ e-def$  by  $blast$ 
  thus  $ts'',move\ ts\ ts''\ v \models \perp$  using  $e-trans\ pcc$  by  $blast$ 
qed
next
case  $False$ 
then have  $neq:d \neq e$  .
show  $False$ 
proof ( $cases\ c=d$ )
  case  $False$ 
  from  $c-def$  have  $ts'',ve \models \langle re(c) \wedge re(e) \rangle$  by  $auto$ 
  hence  $\exists v'. (v' \leq ve) \wedge (ts'',v' \models re(c) \wedge re(e))$ 
    using  $somewhere-leq$ 
    by  $meson$ 
  then obtain  $v'$  where  $v'-def:$ 
     $(v' \leq ve) \wedge (ts'',v' \models re(c) \wedge re(e))$  by  $blast$ 
  with  $backwards-res-stab$  have  $overlap: ts',v' \models re(c) \wedge (re(e))$ 
    using  $c-def\ backwards-res-stab\ c-neq-e\ False$ 
    by ( $metis\ (no-types,\ lifting)\ d-def\ neq$ )
  hence  $unsafe2:ts',ve \models \neg safe(e)$ 
    using  $c-neq-e\ somewhere-leq\ v'-def$  by  $blast$ 
  from  $cr-res.IH$  have  $ts',move\ ts\ ts''\ v \models @e\ (safe(e))$ 
    using  $move-stab$  by  $force$ 
  thus  $False$  using  $unsafe2\ ve-def$  by  $best$ 
next
case  $True$ 
  hence  $e-trans:ts' -r(c) \rightarrow ts''$  using  $d-def$  by  $simp$ 
  from  $c-def$  have  $ts'',ve \models \langle re(c) \wedge re(e) \rangle$  by  $auto$ 
  hence  $\exists v'. (v' \leq ve) \wedge (ts'',v' \models re(c) \wedge re(e))$ 
    using  $somewhere-leq$ 
    by  $meson$ 
  then obtain  $v'$  where  $v'-def:$ 
     $(v' \leq ve) \wedge (ts'',v' \models re(c) \wedge re(e))$  by  $blast$ 
  with  $backwards-res-act$  have  $ts',v' \models (re(c) \vee cl(c)) \wedge (re(e))$ 
    using  $c-def\ backwards-res-stab\ c-neq-e$ 
    by ( $metis\ (no-types,\ lifting)\ d-def\ True$ )

```

```

hence  $\exists v'. (v' \leq ve) \wedge (ts', v' \models (re(c) \vee cl(c)) \wedge (re(e)))$ 
  using v'-def by blast
hence  $ts', ve \models \langle (re(c) \vee cl(c)) \wedge (re(e)) \rangle$ 
  using somewhere-leq move-stab
  by meson
hence  $ts', ve \models \langle re(c) \wedge re(e) \rangle \vee \langle cl(c) \wedge re(e) \rangle$ 
  using hmlsl.somewhere-and-or-distr by blast
thus False
proof
  assume  $assm': ts', ve \models \langle re(c) \wedge re(e) \rangle$ 
  have  $ts', ve \models \neg (c = e)$  using c-def by blast
  thus False using assm' cr-res.IH c-def move-stab ve-def by fastforce
next
  assume  $assm': ts', ve \models \langle cl(c) \wedge re(e) \rangle$ 
  hence  $ts', ve \models \neg (c = e) \wedge \langle cl(c) \wedge re(e) \rangle$  using c-def by blast
  hence  $ts', ve \models \neg (c = e) \wedge \langle cl(c) \wedge (re(e) \vee cl(e)) \rangle$  by blast
  hence  $ts', ve \models pcc\ e\ c$  by blast
  hence  $ts', move\ ts\ ts'\ v \models @e\ (pcc\ e\ c)$ 
    using ve-def move-stab switch-unique by fastforce
  hence  $pcc:ts', move\ ts\ ts'\ v \models (@e\ (pcc\ e\ c)) \vee (@c\ (pcc\ e\ c))$ 
    by blast
  have
     $ts', move\ ts\ ts'\ v \models (\exists d. (@d\ (pcc\ d\ c)) \vee (@c\ (pcc\ d\ c))) \rightarrow \Box r(c) \perp$ 
    using local-LC move-stab c-def e-def by blast
  hence  $ts', move\ ts\ ts'\ v \models \Box r(c) \perp$  using pcc by blast
  thus  $ts'', move\ ts\ ts''\ v \models \perp$  using e-trans by blast
  qed
  qed
  qed
  qed
  qed
  end
  end

```

References

- [1] T. Braüner. *Hybrid logic and its proof-theory*. Springer, 2010.
- [2] M. Hilscher, S. Linker, E. Olderog, and A. Ravn. An abstract model for proving safety of multi-lane traffic manoeuvres. In *ICFEM*, volume 6991 of *LNCS*, pages 404–419. Springer, 2011.
- [3] S. Linker. *Proofs for Traffic Safety: Combining Diagrams and Logic*. PhD thesis, University of Oldenburg, 2015. <http://oops.uni-oldenburg.de/2337/>.

- [4] S. Linker. *Spatial Reasoning About Motorway Traffic Safety with Isabelle/HOL*, pages 34–49. Springer International Publishing, 2017.
- [5] S. Linker and M. Hilscher. Proof theory of a multi-lane spatial logic. *LMCS*, 11(3), 2015.