Modelling and Proving Safety in Autonomous Cars Scenarios in HOL-CSP

Deliverable No. 2 — Final Report

Paolo Crisafulli　　Safouan Taha　　Burkhart Wolff

December 1, 2021

University Paris-Saclay
3 rue Joliot Curie
Bâtiment Breguet
91190 Gif-sur-Yvette

Laboratoire des Methodes Formelles (LMF)
1, Rue Raimond Castaing
91190 Gif-sur-Yvette
France

This document has been generated by Isabelle/DOF version Unreleased/Isabelle2021.

Contributors. We would like to thank the following contributor of this work: Nicolas Méric.

Acknowledgments. This research work has been carried out in the framework of the SVR project (Robot Vehicle Scenarios and autonomous shuttles) at IRT SystemX, Paris-Saclay, France, and therefore granted with public funds within the French Program Investissements d'Avenir (PIA).

Contents

Abstract

We present an approach to model scenarios of autonomous cars in HOL-CSP [10] and prove particular safety properties via interactive proofs in the Isabelle/HOL system (https://en.wikipedia.org/wiki/Isabelle_(proof_assistant)).

The basis of this work is an ontology for Autonomous Car Scenarios given in MOSAR (https://www.mosar.io) that describes a collection of *actors* (e. g. cars, trucks, bicycles), *equipments* (e. g. signals, vehicle lights, etc.), *infrastructures* (e. g. expressways, intersections, etc.) and their dynamic interactions throughout driving scenarios.

We represent the behaviour of actors and (rudimentarily) equipments as *processes*, i. e. infinite sets of traces denoting classes of scenarios. In particular, actors were represented as HOL-CSP processes. Due to the non-determinism and event-polymorphism of HOL-CSP, actor descriptions can be partially defined wrt. to data and arbitrarily "chaotic" in their behaviour. A translation scheme of MOSAR-ontologies into actor processes in HOL-CSP is sketched.

For a particular scenario described in [9] (two cars in a linear line, no backwards driving) we specialize our framework and demonstrate a machine-checked safety proof: If all the actors apply a particular driving strategy taking into account position, speed and acceleration as well as distance to the car in front, there will be no situation with a collision. This strategy — called *Responsibility-Sensitive Safety* — is formulated as a function and the resulting invariant formally proven in Isabelle/HOL, while overcoming a number of short-comings in both the original modeling and the original paper-and-pencil proof.

1 Introduction

This work stands in the context of the *projet SVR (Robot Vehicle Scenarios and autonomous shuttles, https://www.irt-systemx.fr/en/projets/svr/)* affiliated at the *Institut de Recherche Technologique* IRT SystemX (https://www.irt-systemx.fr, Palaiseau, France)

The overall goals of the SVR project are

1. to develop a "common frame of reference" between the branch autonomous bus systems and robot-taxis for both verification and validation of safe systems, where

2. this reference-frame comprises a common language and a library of scenarios serving to construct test-cases in virtual test environments.

   The ISO standard SOTIF (ISO/PAS 21448) [1] introduces a classification of scenarios. The standard introduces a classification of scenarios in several categories called *known safe*, *known unsafe*, *unknown safe* and *unknown unsafe*, depending on whether the scenario is known during the design of the system or discovered during the test phase, and depending on whether the scenario does not destabilize the system or cause it to fail. In the context of the SVR project, it is also a question of defining an approach to list the cases covered and compliant (with success criteria) by the simulation, in order to be able to identify a safe perimeter of use (Operational Design Domain or ODD). In the spirit of the SOTIF standard, this amounts to

   [...] the development of an argument which will demonstrate on the one hand that all the scenarios unknown unsafe is small enough and on the other hand the set of known unsafe scenarios is taken into account through the improvements of SOTIF and therefore that the probability of encountering this type of scenario is low enough. [1, p. 8]

The objective of this sub-project, for which this document is the final deliverable, is to find behavioral models that are sufficiently flexible and open world to take into account *known unsafe* and *unknown unsafe scenarios*, but which are sufficiently well-founded in mathematical logic in order to allow for strong guarantees for (safety) properties for a large set of scenarios, i. e., properties of the form: *for* **all** *scenarios in which cars respect a driving strategy X, there will be no collision.* Note that this work is not about quantifying the probability of encountering *unknown unsafe* scenarios, which remains a question difficult if not impossible to define in the context of an endless number of possible scenarios. Rather, it will be possible to model classes of scenarios allowing simulation, generation of test cases, and / or proof of the inclusion of scenario-classes and relations between behaviours of actors.

The objectives of our sub-project in the SVR context are described as follows:

1. Design sufficiently generic models to represent classes of situations and relevant scenarios.

2. Model domain concepts inspired by SOTIF (such as dangers and situations), ODD (optional), etc.

3. Model scenario classes and study the generation of test cases with coverage criteria to be established.

4. Prove that a class of scenarios is safe, which allows to establish the absence of unknown unsafe scenarios among such a class.

5. Illustrate what an "open world" model based on CSP would be, and its capacity to produce unknown unsafe scenarios.

This list of objectives boils down to two major targets.

The first target is a general modeling framework that allows for giving a common semantics for car scenario modeling languages such as used within the MOSAR platform [1] or Foretellix's M-SDL [2]. The framework will necessarily be very general with respect to expressivity in order to capture both the desired variability of behaviour and data which is expressed in the terms *unknown safe scenarios* and *unknown unsafe scenarios*. A compilation of MOSAR or M-SDL into this common semantic framework should be possible. The framework should support refinement notions in order to structure scenario classes as well as a smooth transition to executable sub-models allowing simulation and test.

Second target of the Analysis: Formalising and proving the safety-property "no collision" for *Responsibility Sensitive Safety* (RSS), a particular driving strategy that controls acceller-ation, speed and distance to the car in front. This represents a concrete instance of the aforementioned general modeling framework which is designed to demonstrate the possibil-ity to formally analyse safety properties by proof techniques inside the framework.

More concretely: we will formalize the concepts of S. Shaliv-Shwartz, S. Shammah, A. Shashua in "On a Formal Model of Safe and Scalable Self-driving Cars"s in the framework and prove formally the intended safety property ("no collision" in *all* situations).

1. We outline the RSS as presented in [9]:
   - A formal model and an analysis of the the collision danger
   - Formal definition of a behaviour (the "driving strategy") compatible with the law ("Duty of Care")
   - Paper-and-pencil proof that this behavior ensures global safety ("Utopia is pos-sible")

2. we provide an instantiation of our general framework for actors as a CSP based model

3. prove formally the re-formulation of the problem as an invariant-preservation proof, and

[1] https://www.mosar.io
[2] https://www.foretellix.com/open-language/

4. prove finally the preservation of this invariant formally in Isabelle/HOL.

A well-known reference theory to study behavioral models is the theory "Concurrent Sequential Processes" (CSP) originally proposed by Anthony Hoare in the late 70ies [4], but has since evolved substantially [2, 3, 8].

Recent work of the authors comprised a formalization of CSP inside Isabelle/HOL [7] and a formal development environment for modeling and theorem proving called HOL-CSP [10]. This environment can cope with events of arbitrary type $'\alpha$, i.e. infinite sets of events, in contrast to model-checking approaches limited to finite ones. This paves the way for models involving processes with rich data states, the handling of dense and real time as well as the handling of newtonian physics by explicit representation of acceleration and speed by vectors in $\mathbb{R}^2$ or $\mathbb{R}^3$. HOL-CSP and an extension module CSP-Ref-Tk (the "CSP Refinement Toolkit") has been archived in the Isabelle "Archive of Formal Proofs" https://www.isa-afp.org in the entries https://www.isa-afp.org/entries/HOL-CSP.html and https://www.isa-afp.org/entries/CSP_RefTK.html and represent therefore the *background* of this work.

Finally, for this report, we use Isabelle/DOF, a Document Ontology Framework, which is able to *enforce* specific ontologies during document evolution by specific Isabelle IDE support. Thus the *coherence* between the formal and the informal parts of the content of this document can be mechanically checked.

2 The CSP-based Generic Autonomous Car Model

The setup of this Core-Theory results from importing the following background theories:

```
theory CSP−AutoCars
 imports Isabelle_DOF.technical_report
        CSP_RefTK.Properties  CSP_RefTK.CSP_ext
        HOL.Real_Vector_Spaces HOL−Analysis.Product_Vector
```

Beyond technicalities for the underlying document preparation, this core theory is based on the following library components:

1. Vector-Spaces and analysis packages for modeling physical quantities, and
2. CSP and the CSP-Refinement Toolkit in order to model situations (snapshots), scenarios (a sequence of snapshots) and relations between classes of scenarios.

2.1 Fundamental Modeling Assumptions

We will use *parametric polymorphism* [1] in order to model the class hierarchies on actors (resp. their states) and their extensibility. Since actor states will comprise physical quantities like position, speed, acceleration, the global world state based upon the states of all its actors (and environment components), the latter will also be polymorphic and extensible.

We chose to reuse the theory *HOL.Real_Vector_Spaces* from the Isabelle/HOL library in order to model the above mentioned quantities as vectors. This theory provides the type-class *real_normed_vector*, which provides the algebraic structure of vector-spaces for types such as $\mathbb{R}$, $\mathbb{R}\times\mathbb{R}$, $\mathbb{Q}^3$, or even machine type models such as, for example, the 2-dimensional *64 word* $\times$ *64 word* over 64-bit-vectors as occuring in concrete calculation in a programming language.

The only compromise that this choice implies is that *real_normed_vector* comes with a scalar multiplication [2] $a *_R x$ for which the a is required to be a real number $\mathbb{R}$. In order to model the physics of vector spaces in Newtonian physics, however, this choice is utmost nearly unavoidable: real-valued scaling factors appear naturally for vectors in physical computations such as $(1::'a)\ /\ 2 *_R t^2 *_R a$ or even $sqrt\ 2 *_R a$ (as occurring naturally in norms of 2-dimensional vectors).

[1] https://en.wikipedia.org/w/index.php?title=Parametric_polymorphism&oldid=1038893302

[2] https://en.wikipedia.org/w/index.php?title=Scalar_multiplication&oldid=1039985947

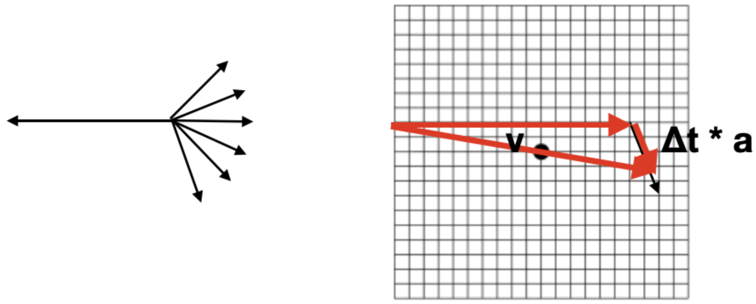


Figure 2.1: Generic Vector-Spaces in the Autonomous Car Modeling Framework.

2.2 Actor States and its Extensions

We model the *dynamic environment* of an autonomous car *ego* as a system of parallel processes. These processes have a state — a position, a speed, a set of possible positive and negative accelerations, potentially a physical extension — modeling the physical quantities of these objects. Speed can only evolve via acceleration over time, and positions must be bound to a topology. However, positions, speeds and accelerations are parameterized in this model and can be (two-dimensional, three-dimensional) vectors in discrete or non-discrete space-times. The mother of all actor states is thus parameterized by vector-type-parameter $'v$ which is constrained to the type-class *real_normed_vector*.

record $('v$::*real_normed_vector*$)$ *as* = — for actor state
  pos :: $'v$ — current position
  speed :: $'v$ — current speed
  acc :: $'v$ — current acceleration

Based on parametric polymorphism, it is possible to express some form of single-inheritance as required by our overall objective *"common frame of reference"* which should be extensible and therefore allow forms of *"unknowns"* in the models.

The trick is done by representing **record** types as cartesian products $\tau_1 \times ... \times \tau_n \times \alpha$ where the τ_i correspond to the types of the attributes and the type-variable α stands for an "extension field" of the record. In this view, the "attributes" of a record become a projection function into the cartesian tuple.

For example, the **record** representation above for the most general actor state space can be seen as a product type $'v \times 'v \times 'v \times \alpha$ with the projection functions:

1. *pos* $\equiv$ *fst*
2. *speed* $\equiv$ *fst o snd*
3. *acc* $\equiv$ *fst o snd o snd*

4. and the implicit projection on the *more*-field: *more* $\equiv$ *fst o snd o snd o snd* (where *fst* and *snd* are the usual projections into a binary cartesian product and *o* is the function composition.)

   Isabelle/HOL generates from a record declaration a special syntax for its support. Rather than tuple notations such as (*a*,*b*,*c*,*more*), records can be denoted by ⦇ *pos* = *a*, *speed* = *b*, *acc* = *c*, ... = *more* ⦈, and the corresponding projection functions were defined to work directly on this notation. Isabelle also introduces a notation for record-types which were written $('v,\ '\alpha)$ *as_scheme* for which the synonym $('v,'a)$ *as_scheme* is generated.

   Note that Isabelle also supports the equivalent notation: ⦇ *pos* = *a*, *speed* = *b*, *acc* = *c* ⦈ for ⦇ *pos* = *a*, *speed* = *b*, *acc* = *c*, ... = () ⦈. Here, the () is the "empty-product", i. e. the only element in the so-called *unit*- type. This corresponds to the idea of a *finalized* class in an class hierarchy, i. e. a class from which no further inheritance is possible (and which is therefore "known" in all respects).

   Isabelle can prove theorems over states containing polymorphic variables, that is, that will hold for all type instances of this theorem. This implies, that theorems can be established on schematic actor states $('v,'a)$ *as_scheme* that hold for all future extensions, thus holds over a certain form of "unknowns".

   However, we have to be precise if we want to reason over a closed state-space (so: *'v as* or equivalently: *'v as*) or an open state-space: (so: $('v,\ '\alpha)$ *as_scheme* or equivalently: $('v,\ '\alpha)$ *as_scheme*). In order to bridge a little the gap between these two forms we introduce by convention the following **type_synonym**'s for each actor-state class:

type_synonym $('v,'\alpha)$ *as_* = $('v,'\alpha)$ *as_scheme*

Examples of Actor Instances:

term ‹*pos* :: ('*v*::*real_normed_vector*, $'\alpha$) *as_* ⇒ '*v*›
term‹ ⦇ *pos* = *14*, *speed* = *2*, *acc* = −*1* ⦈ :: *real as*›
— One-dimensional *real as*.
term‹ ⦇ *pos* = (*14*, *15*), *speed* = (*sqrt 2*, *2.4*),
acc = (*0*, *0*), ... = *a* ⦈ :: (*real* × *real*, $'\alpha$) *as_*›
— Two-dimensional, extensible actor state space of type (*real* × *real*, $'\alpha$) *as_*.

We provide the first major extension of the actor state space: we characterize an actor by a set of possible accelerations it may choose from, and a physical extension as an object which becomes relevant in case of collisions:

record ('*v*::*real_normed_vector*) as_{range} = ‹'*v as*› +
acc_range :: ‹'*v set*› — The set of possible accelerations of an actor. Depending on the instance for the underlying vector-space, this can be an acceleration in front, towards a side, or braking.
extension_field :: ‹'*v set*› — gives the expansion of the physical object in terms of a set of positions (relative vectors).

Again, following our convention, we provide an abbreviation for the "open" version of this actor state space:

```
type_synonym ('v,'a) as_range_ = ('v,'a) as_range_scheme
```

This construction implies the need of a particular invariant for $('v, '\alpha)$ as_{range}_: the current acceleration must be one of the possible ones:

```
definition wd_as:: ‹('v::real_normed_vector,'a) as_range_ ⇒ bool›
  where ‹wd_as σ ≡ acc σ ∈ acc_range σ›
corollary ‹  wd_as (| pos = (14::real), speed = 2,
                   acc = −1, acc_range = {−1..1}, extension_field = {}|)›
    and ‹¬ wd_as (| pos = (14::real), speed = 2,
                   acc = −1, acc_range = {0..1}, extension_field = {} |)›
  by (simp_all add:wd_as_def)
```

2.3 Global States: Scenes and "Open Scenes"

Scenes are global system states comprising actor states.

```
type_synonym id_actor        = nat — an index for actors
type_synonym 'v scene        = ‹id_actor ⇒ 'v as›
                              — a scene where all objects are identified by their index
type_synonym ('v,'α) scene_ = ‹id_actor ⇒ ('v,'α) as_›
                              — ... the "open" version
```

Note that the above model of an "Open Scene" is not that open after all: while it encompasses the possibility of completely new actors and their characteristics, it does not allow open-ness wrt. *equipments* (e. g. signals, vehicle lights, etc.) and *infrastructures*. However, we can apply the general extension techniques of data in Isabelle/HOL also for this objective:

```
record ('v,'α)scene_o =  global_state :: id_actor  ⇒ ('v,'α)  as_range_
                 — a record with a single field named global_state
type_synonym ('v,'α,'β)scene_o_ = ('v,'α,'β)scene_o_scheme
                 — and its open version
```

Note that the demon-state is still extensible: we can add a new table that captures the state of environment components such as traffic lights or fences/gates etc.

Let ω be a *'v scene*. Then we provide a predicate $no_{collision}$ ω which is true when all actors have relative distances ε.

```
definition no_collision::‹('v::real_normed_vector) scene ⇒ id_actor set ⇒ 'v ⇒ bool›
  — when actors should keep some ε as distance from each other
  where ‹no_collision ω sid ε ≡ ∀ i∈sid. ∀ j∈sid. norm (pos(ω j)−pos(ω i)) ≤ norm ε ⟶ i = j›

corollary no_collision_ε_0: ‹no_collision ω sid 0 = inj_on (λi. pos (ω i)) sid›
  — when actors are points we can set ε to zero
  unfolding inj_on_def no_collision_def by auto
```

2.4 Motions (Driving Strategies)

Let us define motions as descriptions of the moving behavior of the actors, or, in other words, as a driving strategy giving next states after taking into account the capture of the surrounding world.

type_synonym time = real
type_synonym 'v motion = ‹id_{actor} ⇒ time ⇒ 'v scene ⇒ 'v as set›
— function returning the new set of states for a given actor, given an actor id, a δt and an initial scene.
type_synonym ('v,'α) motion_ = ‹id_{actor} ⇒ time ⇒ ('v,'α) scene_ ⇒ ('v,'α) as_ set›
— ... and the "open" version.

lemma motion_step:
assumes ‹∀ i∈sid. motion i δt ω ≠ {}›
— assume motion returns no empty sets of actor states on *sid*
shows ‹∃ω'. ∀ i∈sid. ω' i ∈ motion i δt ω›
— then there is some scene ω' corresponding to the result (image set) of a motion
by (exI ‹λi. (SOME s. s ∈ motion i δt ω)›)
— this λ returns some actor state for every actor id: it is a scene.
— hence, we have built a scene, showing the lemma: *qed*
(simp add: assms some_in_eq)

2.5 CSP Model I

2.5.1 Preliminaries

declare cont_fun[simp] — missing in the general reasoning on continuity.

2.5.2 Events (Simplified)

datatype 'v events = sync (dt: real)(sc: ‹'v scene›) — simulation step
| up ‹'v scene› — update

abbreviation ‹$sync_{pair}$ ≡ λ(δt,ω). sync δt ω›
— another way to denote the constructor *events.sync*

— proof tactic to break the cartesian product, not relevant
lemma $sync_{pair}$_simp[simp]:
‹Mprefix (range $sync_{pair}$) (P ∘ inv $sync_{pair}$) = Mprefix (range $sync_{pair}$) (λe_{sync}. P (dt e_{sync}, sc e_{sync}))›
by (smt (z3) case_prod_beta' comp_apply events.sel(1) events.sel(2) f_inv_into_f mprefix_eq prod.collapse)

2.5.3 The Behaviour of Actors as CSP-Processes

The processes generated by a set of actors and a motion function they share:

definition μ_{motion} :: ‹id_{actor} set ⇒ 'v motion ⇒ ('v::real_normed_vector) events process› (⟦(_):(_)⟧)
where ‹μ_{motion} sid motion ≡ μ X. $sync_{pair}$**?**(δt,ω) → (up!ω'|(∀ i∈sid. ω' i ∈ motion i δt ω) → X)›

The "usual" recursive lemma, used for simplification and fixpoint reasoning:

lemma μ_{motion}_rec: ‹⟦sid:motion⟧ = $sync_{pair}$**?**(δt,ω)

```
                              → (up!ω′|(∀ i∈sid. ω′ i ∈ motion i δt ω)
                              → ⟦sid:motion⟧)›
  by (simp add:μ_motion_def, subst fix_eq, simp)
```

This lemma allows to reduce CSP refinement proofs to "denotational" ones: if motion 1 is included in motion 2, then CSP1 refines CSP2

```
lemma μ_motion_refine:
  assumes a: ‹∀ δt ω i. (i ∈ sid ∧ motion₂ i δt ω ≠ {}) ⟶ (motion₁ i δt ω ≠ {})›
    and b: ‹∀ δt ω i. i ∈ sid ⟶ motion₁ i δt ω ⊆ motion₂ i δt ω›
  shows ‹⟦sid:motion₂⟧ ≤ ⟦sid:motion₁⟧›
proof(subst μ_motion_def, induct rule: fix_ind)
  show ‹adm (λa. a ≤ (⟦sid:motion₁⟧))› by (simp_all add: monofunI)
next
  show ‹⊥ ≤ ⟦sid:motion₁⟧› by simp
next
  case induct_case:(3 x)
  have c:‹   (⊓e∈up ‘ {ω′. ∀ i∈sid. ω′ i ∈  motion₂ i δt ω} → x)
          ≤  (⊓e∈up ‘ {ω′. ∀ i∈sid. ω′ i ∈ motion₁ i δt ω} →  x)› for δt ω
    apply (cases ‹∃ i∈sid. motion₂ i δt ω = {}›)
    apply (metis (no_types, lifting) Collect_empty_eq b bot.extremum_uniqueI empty_iff eq_iff)
    apply(rule Mndetprefix_FD_subset) apply (auto simp:a) apply(exI ‹λi. SOME s. s∈motion₁ i
δt ω›)
    apply (simp add: a some_in_eq)
    using b by blast
  show ?case
    apply(subst μ_motion_rec, simp add:c read_def)
    apply(rule mono_mprefix_FD)
    using order_trans[OF c] by (meson induct_case.hyps mono_Mndetprefix_FD)
qed
```

2.5.4 Maxwells Daemon (I)

In classical physics, Maxwell's demon is a concept around thought-experiments that postulate the existence of a kind of actor that "knows" the global state of all physical objects at a given point in time. Tongue-in-cheek, we will postulate this demon here as well (not diving in the discussion about the possibility of its existence in thermo-dynamics); one can see it as an "orchestrator" or "mediator" in our modeling framework.

In our framework, we assume a "Maxwell's demon":

1. that chooses non-deterministically some positive real sampling interval δt strictly smaller that some bound Δt
2. that "knows" for given points in time the scene (global physical state) σ of all actors.

Actors (and in future extensions of the model, similarly: infrastructure components):

1. receive the δt from the demon,

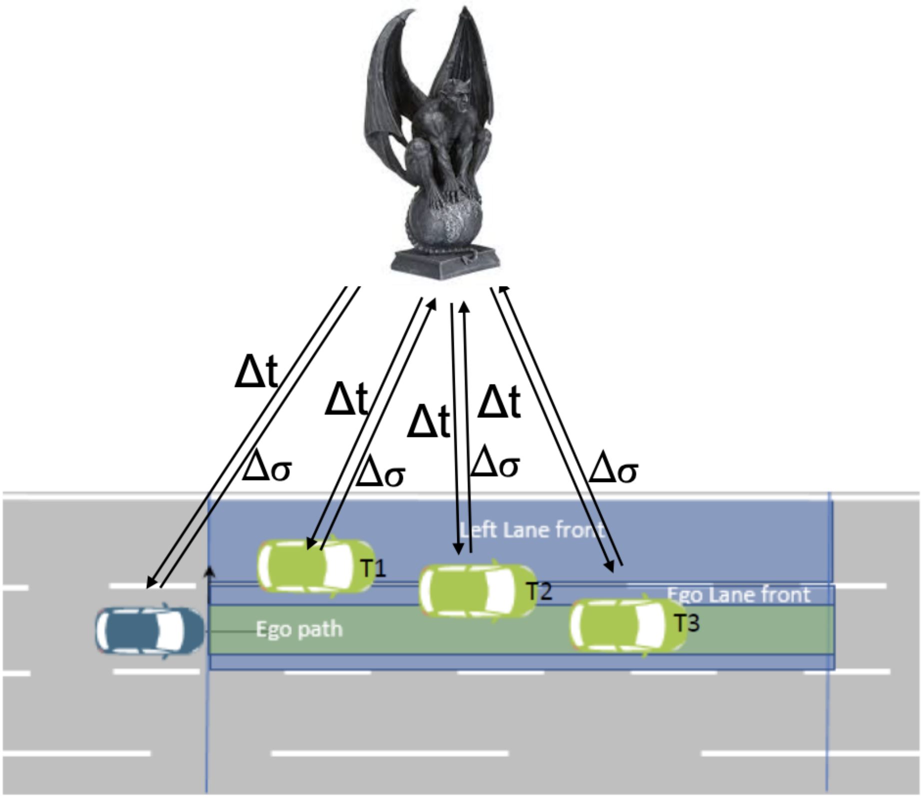


Figure 2.2: The Behaviour of the Orchestrator (The "Maxwell Demon").

2. send their physical state $\Delta\sigma$ to him,
3. receive the scene (global state σ),
4. decide non-deterministically their possible accelerations according to their driving strategy which is sensitive to the road/environment (topology) and the scene σ, and
5. are sensitive in their choice via their driving strategy to the scene σ.

In this section, we present a somewhat simplified demon which makes strong assumptions on synchronisation over the global scene — basically 1) all actors learn about the global situation via a kind of broadcast and 2) the absence of infrastructure elements such as signals and communication between actors — in order to simplify the major proofs in the proof chapters over different versions of RSS driving strategies. A version of the *demon* overcoming these limitations is presented in the next chapter.

definition *demon* :: ‹$real \Rightarrow ('v::real\_normed\_vector)\ scene \Rightarrow 'v\ events\ process$›
where ‹$demon\ \Delta t \equiv \mu X. (\lambda\ \omega. (sync_{pair}!(\delta t, \omega_s)|(\delta t \in \{0{<}..\Delta t\} \wedge \omega_s = \omega) \rightarrow (up?\omega' \rightarrow X\ \omega')))$›

Here, $\{0::'a{<}..\Delta t\}$ denotes the set containing the interval beween 0 (exclusively) and a given clock-time Δt.

Let's do some formal verification : let's prove the "usual" recursive unfolding lemma...

lemma *demon_rec*:
‹$demon\ \Delta t\ \omega = (sync_{pair}!(\delta t, \omega_s)|(\delta t \in \{0{<}..\Delta t\} \wedge \omega_s = \omega) \rightarrow (up?\omega' \rightarrow demon\ \Delta t\ \omega'))$›
by (*simp add:demon_def*, *subst fix_eq*, *simp*)

Note that the ($\sqcap$)-operator from CSP lets the demon choose an arbitrary time interval smaller than some bound Δt and send it to all actors, which respond with their actor state after that time interval.

2.5.5 Safe Scenarios

We define here a *scenario* as the constructor of a process, based on a set of actor ids, a motion (driving strategies of the actors), a maximum time step and an initial scene. The resulting process is the sequence of simulation steps (*sync* events) triggered by the demon.

definition *scenario* ::
‹$id_{actor}\ set \Rightarrow ('v::real\_normed\_vector)\ motion \Rightarrow real \Rightarrow 'v\ scene \Rightarrow 'v\ events\ process$›
where ‹$scenario\ sid\ motion\ \Delta t\ \omega_0 \equiv (demon\ \Delta t\ \omega_0 \mid\mid [\![sid{:}motion]\!]) \setminus \{e.\ \neg\ is\_sync\ e\}$›

A safe (instantiated) scenario must refine *safe* (once instantiated with actor ids and an ε value).

definition *safe* :: ‹$id_{actor}\ set \Rightarrow ('v::real\_normed\_vector) \Rightarrow 'v\ events\ process$›
where ‹$safe\ sid\ \varepsilon \equiv \mu X. (sync_{pair}!(\delta t, \omega_s)|(\delta t > 0 \wedge no_{collision}\ \omega_s\ sid\ \varepsilon) \rightarrow X)$›

definition is_{safe} **where** ‹$is_{safe}\ sid\ motion\ \Delta t\ \omega_0\ \varepsilon \longleftrightarrow safe\ sid\ \varepsilon \leq scenario\ sid\ motion\ \Delta t\ \omega_0$›

2.5.6 Scenario Refinement

monotony ofis_{safe}: stricter driving strategies inherit safety from less rigorous ones ...

```
lemma is_safe_refine:
  assumes a:‹⟦sid:motion₂⟧ ≤ ⟦sid:motion₁⟧›
    and b:‹is_safe sid motion₂ Δt ω₀ ε›
   shows ‹is_safe sid motion₁ Δt ω₀ ε›
proof -
  have ‹scenario sid motion₂ Δt ω₀ ≤ scenario sid motion₁ Δt ω₀›
    by (simp add:scenario_def a)
  then show ?thesis
    by (meson b is_safe_def order_trans)
qed


lemma scenario_rec:
‹scenario sid motion Δt ω = (⊓e_sync∈{sync δt ω |δt. δt ∈ {0<..Δt}}
                              → (⊓ω'∈{ω'. ∀ i∈sid. ω' i ∈ motion i (dt e_sync) ω}.
                                    scenario sid motion Δt ω'))›
  apply(subst scenario_def, subst demon_rec, subst μ_motion_rec)
  apply(simp add:read_def, subst gndet_mprefix_sync) apply (simp add: image_mono)
  apply(subst sync_commute, subst gndet_mprefix_sync) apply (simp add: image_mono)
  apply (subst Mndet_Hide1) apply fastforce
  apply (subst Mndet_Hide2) apply fastforce


proof(goal_cases)
  case 1
  have a:‹sync_pair ` {(δt, ω_s). 0 < δt ∧ δt ≤ Δt ∧ ω_s = ω}
        = {events.sync δt ω | δt. 0 < δt ∧ δt ≤ Δt}› by auto

  have b:‹(inv events.up (events.up x)) = x› for x::‹nat ⇒ 'v as›
    by (metis events.inject(2) f_inv_into_f rangeI)
  show ?case

    apply(subst a)
    apply(intro mndet_eq allI impI, auto, rule gndet_eq2, auto, subst b)
    by (simp add: par_comm scenario_def)
qed
```

Let ω_0 be a scene with no collision, *motion* a driving strategy always mapping to a scene with no collision, hence the corresponding scenario is safe

```
lemma is_safe_rule1:
  assumes a:‹Δt > 0›
    and b:‹no_collision ω₀ sid ε›
    and c:‹⋀ω δt. no_collision ω sid ε ∧ δt ∈{0<..Δt}
                ⟹ let A = {ω'. ∀ i∈sid. ω' i ∈ motion i δt ω}
                    in A ≠ {} ∧ (∀ω'∈A. no_collision ω' sid ε)›
   shows ‹is_safe sid motion Δt ω₀ ε›
```

$no_{collision}$?ω sid ε ∧ ?δt ∈ {0<..Δt} ⟹ let A = {ω'. ∀ i∈sid. ω' i ∈ motion i ?δt ?ω} in A ≠ {} ∧ (∀ ω'∈A. $no_{collision}$ ω' sid ε) defines *motion* such that starting from a collision free scene, we move to another collision free scene

```
using b proof(unfold is_safe_def safe_def, induct arbitrary:ω0 rule:fix_ind)
  case 1
  then show ?case by (simp add:monofun_def)
next
  case 2
  then show ?case by simp
next
  case (3 x)
  have d:‹⊓_∈sync_pair ‘ (Collect ((<) 0) × {ω_s. no_collision ω_s sid ε}) → x ≤
        ⊓e_sync∈{events.sync δt ω0 |δt. 0 < δt ∧ δt ≤ Δt} → x›
    apply(rule Mndetprefix_FD_subset)
    using 3 a c by auto
  have ‹δt ∈{0<..Δt} ⟹ x ≤ gndet {ω'. ∀ i∈sid. ω' i ∈ motion i δt ω0} (scenario sid motion Δt)›

      for δt
    apply(rule mono_gndet_const)
    using 3 c by auto
  with 3 show ?case
    apply(subst scenario_rec, auto)
          apply(intro trans_FD[simplified failure_divergence_refine_def, OF d],rule
mono_Mndetprefix_FD)
    by auto
qed
```

— a second variant lemma, strictly equivalent to is_{safe}_rule1

```
lemma is_safe_rule2:
  assumes a:‹Δt > 0›
      and b:‹no_collision ω0 sid ε›
      and c:‹⋀ω δt. no_collision ω sid ε ∧ δt ∈{0<..Δt}
                  ⟹ ∀ i∈sid. motion i δt ω ≠ {}›
      and d:‹⋀ω δt. no_collision ω sid ε ∧ δt ∈{0<..Δt}
                  ⟹ ∀ ω'∈{ω'. ∀ i∈sid. ω' i ∈ motion i δt ω}. no_collision ω' sid ε›
    shows ‹is_safe sid motion Δt ω0 ε›
by (intro is_safe_rule1 a b, auto simp:c d motion_step)


lemma is_safe_rule_inv:
  fixes P_inv::‹(nat ⇒ ('v::real_normed_vector) as) ⇒ nat set ⇒ bool›
  — let P_inv be a predicate
  assumes a:‹Δt > 0›
      and b:‹P_inv ω0 sid›
      — assume P_inv is true for scene ω0
      and c:‹⋀ω δt. P_inv ω sid ∧ δt ∈{0<..Δt} ⟹ ∀ i∈sid. motion i δt ω ≠ {}›
      — given a non empty motion
      and d:‹⋀ω δt. P_inv ω sid ∧ δt ∈{0<..Δt}
                  ⟹ ∀ ω'∈{ω'. ∀ i∈sid. ω' i ∈ motion i δt ω}. P_inv ω' sid›
```

```
    — which preserves P_inv
    and e:‹⋀ω. P_inv ω sid ⟹ no_collision ω sid ε›
    — assume P_inv implies no collision
  shows ‹is_safe sid motion Δt ω_0 ε›
    — then the motion is safe
  using b
proof(unfold is_safe_def safe_def, induct arbitrary:ω_0 rule:fix_ind)
  show adm (λa. ∀x. P_inv x sid ⟶ P_inv x sid ⟶ a ≤ scenario sid motion Δt x)
    by (simp add:monofun_def)
next
  fix ω_0::nat ⇒ 'v as
  assume P_inv ω_0 sid and P_inv ω_0 sid
  then  show ⊥ ≤ scenario sid motion Δt ω_0
    by simp
next
  fix x fix ω_0::nat ⇒ 'v as
  assume hyp: (⋀ω_0. P_inv ω_0 sid ⟹ P_inv ω_0 sid ⟹ x ≤ scenario sid motion Δt ω_0)
    and  base: P_inv ω_0 sid
    have d:‹⊓_∈sync_pair ‘ (Collect ((<) 0) × {ω_s. no_collision ω_s sid ε}) →
        x  ≤ ⊓e_sync∈{events.sync δt ω_0 |δt. 0 < δt ∧ δt ≤ Δt} → x›
        apply(rule Mndetprefix_FD_subset) using a e[of ω_0, OF base] by auto

    have e:‹δt ∈{0<..Δt}
      ⟹ x ≤ gndet {ω'. ∀i∈sid. ω' i ∈ motion i δt ω_0} (scenario sid motion Δt)› for δt
      apply(rule mono_gndet_const)
      using base hyp c assms(4) apply auto by (rule motion_step, simp)
  with e
  show (Λ x. ⊓_∈sync_pair‘{(δt, ω_s). 0<δt ∧ no_collision ω_s sid ε} → x)·x ≤ scenario sid motion
Δt ω_0
    apply(subst scenario_rec, auto)
          apply(intro trans_FD[simplified failure_divergence_refine_def, OF d], rule
mono_Mndetprefix_FD)
    by auto
qed

— computing a motion from a set of accelerations following Newtonian kinetics: motion by acc
type_synonym 'v motion_acc = ‹id_actor ⇒ time ⇒ 'v scene ⇒ 'v set›
      — the set of possible accelerations for a given actor, δt and initial scene
definition kinetics :: ‹('v::real_normed_vector) motion_acc ⇒ 'v motion›
  where  ‹kinetics motion_acc ≡
      — defines a motion for a δt such that a (acceleration) remains constant during δt
        λ i δt ω. { ⦇ pos   = pos (ω i) + (δt) *_R  speed (ω i) + ((δt^2)/2) *_R a,
                      speed = speed (ω i) + (δt *_R a),
                      acc   = a ⦈
                    |a. a ∈ motion_acc i δt ω}›

lemma kinetics_empty: ‹kinetics motion_acc i δt ω = {} ⟷ motion_acc i δt ω = {}›
  by (auto simp add: kinetics_def)
```

```
lemma μkinetics_refine:
  assumes a: ‹∀ δt ω i. (i ∈ sid ∧ accs2 i δt ω ≠ {}) ⟶ (accs1 i δt ω ≠ {})›
      and b: ‹∀ δt ω i. i ∈ sid ⟶ accs1 i δt ω ⊆ accs2 i δt ω›
      — assume motionacc1⊆motionacc2
    shows ‹⟦sid:kinetics accs2⟧ ≤ ⟦sid:kinetics accs1⟧›
      — then the corresponding kinetics generate refining processes
  apply(intro allI impI μmotion_refine)
  using a b by (auto simp add:kinetics_def) blast


— obvious case: no move or same move no collision
declare inj_on_def[simp] One_nat_def[simp del]

theorem no_acc_is_safe:
  fixes noacc::‹('v::real_normed_vector) motionacc›
  assumes a:‹Δt > 0›
      and b:‹noacc ≡ λ i δt ω. {0}›
      and c:‹nocollision ω0 {0..<n} ε ∧ (∀ i<n. speed (ω0 i) = V)›
    shows ‹issafe {0..<n} (kinetics noacc) Δt ω0 ε›
proof -
  define Pinv::‹(nat ⇒ ('v::real_normed_vector) as) ⇒ nat set ⇒ bool›
    where ‹Pinv ≡ λ ω sid. nocollision ω sid ε ∧ (∀ i∈sid. (speed::'v as ⇒ 'v) (ω i) = V)›
  show ?thesis
    apply(rule issafe_rule_inv[where Pinv=Pinv], simp_all add:assms Pinv_def kinetics_def)
    using c nocollision_def by (metis add_diff_cancel_right as.select_convs(1))
qed

theorem same_acc_is_safe:
  fixes sameacc::‹('v::real_normed_vector) motionacc› and A::'v
  assumes a:‹Δt > 0›
      and b:‹⋀δt ω. ∃ a. ∀ i. sameacc i δt ω = {a}›
      and c:‹nocollision ω0 {0..<n} ε ∧ (∀ i<n. speed (ω0 i) = V)›
    shows ‹issafe {0..<n} (kinetics sameacc) Δt ω0 ε›
proof -
  define Pinv::‹(nat ⇒ ('v::real_normed_vector) as) ⇒ nat set ⇒ bool›
    where ‹Pinv ≡ λ ω sid. nocollision ω sid ε ∧ (∀ i j. i∈sid ∧ j∈sid ⟶ (speed (ω i) = speed (ω
j)))›
  show ?thesis
  proof(rule issafe_rule_inv[where Pinv=Pinv], simp_all add:assms Pinv_def, goal_cases)
    case (1 ω δt)
    then show ?case using c by (metis b empty_not_insert kinetics_empty)
  next
    case (2 ω δt)
    then show ?case
      unfolding nocollision_def kinetics_def
     by (smt (z3) add_diff_cancel_right as.select_convs(1) as.select_convs(2) atLeastLessThan_iff

                   b linorder_not_less mem_Collect_eq not_less_zero singletonD)
  qed
```

```
qed

end
```

3 From MOSAR-Ontologies to a Semantics in CSP

```
theory   MOSAR−Encodings
 imports CSP−AutoCars

begin
```

3.1 More Examples: Expressing the MOSAR hierarchy of Actor States

```
term acc_range (X::('v::real_normed_vector,'b) as_range_)

record  ('v::real_normed_vector) Usual_Road_User_σ = 'v as_range +
      speed_max  :: real
      passengers :: nat

type_synonym ('v,'a) Usual_Road_User_σ_ = ('v,'a) Usual_Road_User_σ_scheme


record ('v::real_normed_vector) PassengerCar_σ = 'v Usual_Road_User_σ +
      xxx :: string

type_synonym ('v,'a) PassengerCar_σ_ = ('v,'a) PassengerCar_σ_scheme


record ('v::real_normed_vector) Bike_σ = 'v Usual_Road_User_σ +
      yyy :: string

type_synonym ('v,'a) Bike_σ_ = ('v,'a) Bike_σ_scheme


record ('v::real_normed_vector) Truck_σ ='v Usual_Road_User_σ +
      zzz :: string

type_synonym ('v,'a) Truck_σ_ = ('v,'a) Truck_σ_scheme
```

The moving object types can be hierarchically extended — space for "unknown" features and its properties.

3.1.1 Examples

One-dimensional *int as*.

```
term‹ person = ⦇ pos = 14,
                 speed = 2,
                 acc = −1,
                 acc_range = {−2,−1,0,1},
                 extension_field = {1, 0, −1} ⦈›
```

Two-dimensional (*rat* × *rat*) *as*.

```
term‹ (1.0, 1.0) :: ((rat × rat))›
```

```
term‹ racing_car = ⦇ pos = (140.0,210.0)::rat×rat,
                     speed = (200.0,0.0),
                     acc = (10.0,0.0),
                     acc_range = {(10.0,0.0),    — acceleration ahead
                                  (2.0,3.0),     — acceleration to left
                                  (2.0,−3.0),    — acceleration to right
                                  (−40.0,0.0),   — halfbreak
                                  (−80.0,0.0)},  — fullbreak
                     extension_field = {(x,y). x∗x + y∗y ≤ 4.0} ⦈›
                                        — kis: disk with 4 units diameter.
```

```
definition collision :: ('v::real_normed_vector,'a) as_range_
                         ⇒ ('v,'a) as_range_
                         ⇒ bool (infix 70)
  where emo1 emo2 ≡ (∀ e1 ∈ (extension_field emo1).
                        ∀ e2 ∈ (extension_field emo2).
                          ((pos emo1) + e1 ≠ (pos emo2) + e2 ))
```

```
definition dynamics :: real
                        ⇒ (('v::real_normed_vector), 'a) as_range_
                        ⇒ ('v,'a) as_range_ set
  where dynamics δt σ ≡ {σ' . ∃ acc ∈ acc_range σ.
                            let speed' = speed σ + (δt ∗R acc)
                            in σ' = σ⦇speed := speed',
                                       pos := pos σ +
                                              (0.5 ∗R δt ∗R (speed σ + speed'))⦈
                        }
```

```
definition pos_speed :: ('v::real_normed_vector,'a) Usual_Road_User_σ_ set
  where pos_speed ≡ {σ. 0 ≤ norm(speed σ) ∧ norm(speed σ) ≤ speed_max σ }
```

3.2 Static Environment : The Street Topology

datatype $lane\_type = sidewalkL \mid boardwalkzoneL$
$\mid laneL1 \mid midlane12 \mid laneL2$
$\mid boardwalkzoneR \mid sidewalkR$

type_synonym $'v\ topology = lane\_type \Rightarrow 'v\ set$

3.2.1 Linear Lane Example

definition $linear\_lane \equiv undefined\ (laneL1 := \{0\ ..\ 100::int\})$

3.3 CSP Model II

In this section, we present a more refined demon which makes weaker assumptions on synchronisation over the global scene and allows communication of infrastructure elements such as signals and communication between actors.

3.3.1 Generalized Events

— Moving objects signal their status as moving object
datatype $('v,'\alpha,'\beta)\ events = up_{local}\ (id : id_{actor})$
$(get\_status : ('v,'\alpha)\ as_{range}\_)\ (\langle\langle\_, \_\rangle\rangle)$
$\mid sync\ (world\_state : ('v,'\alpha,'\beta)scene_{o}\_)$
$\mid time\_lapse\ (time : real)$

term $\langle d?(y,y) \rightarrow c!x \rightarrow P = Q \rangle$

definition $\langle status_c = case\_prod\ up_{local} \rangle$ — necessary for uncurried notation in read commands

term $\langle status_c?(ide,\sigma) \rightarrow X((ds::id_{actor} \rightharpoonup ('v,'a)\ as_{range}\_)\ (ide \mapsto \sigma)) \rangle$

3.3.2 Maxwells Daemon II

In classical physics, Maxwell's demon is a concept around thought-experiments that postulate the existence of a kind of actor that "knows" the global state of all physical objects at a given point in time. Tongue-in-cheek, we will postulate this demon here as well (not diving in the discussion about the possibility of its existence in thermo-dynamics); one can see it as an "orchestrator" or "mediator" in our modeling framework.

definition $demon :: real \Rightarrow (id_{actor} \Rightarrow ('v::real\_normed\_vector,'\alpha)\ as_{range}\_) \Rightarrow ('v,'\alpha,'\beta)$ $events\ process$
where $\langle demon\ \Delta t \equiv \mu\ X.\ (\lambda\ ds.\ \sqcap\ \Delta t \in \{time\_lapse\ x\ |x.\ 0 < x \wedge x \leq \Delta t\}$
$\rightarrow (\mu\ X.\ (\lambda\ ds.\ status_c?(ide,\sigma) \rightarrow (X\ (ds(ide := \sigma)))))\ ds$
$\sqcap\ X\ ds$ — inner loop for reads; not necessarily df
$)\rangle$
term $\langle\ info?\sigma_G \rightarrow P \rangle$

3.3.3 Behavior of Actors

```
definition μas :: (('v,'α,'β)sceneo_ ⇒ ('v, 'α) asrange_ set) ⇒ nat ⇒ ('v, 'α) asrange_
                ⇒ ('v::real_normed_vector,'α,'β) events process
  where ‹μas driving_strategy ide
          ≡ μ X. (λ as. □ Δt ∈ {time_lapse x |x. 0 ≤ x}
                     → sync?σG
                       → (⊓ as' ∈ (uplocal ide)`((dynamics(time Δt)as)∩driving_strategy σG)
                           → (X (get_status as'))))›

syntax _asbeh :: ('v, 'a) as_ ⇒ ('v, 'a) as_ set ⇒ nat
                 ⇒ ('v::real_normed_vector,'a,'b) events process ([[(_)|(_)]]_)

translations [[as | A ]]n ⇌ (CONST μas) A n as
```

3.3.4 Examples

```
definition rear_car ≡ 0::idactor
definition front_car ≡ 1::idactor
definition mt x ≡ undefined


definition ‹racing_car = undefined›
definition ‹on_lane1 = undefined›

definition ego_id = (0::idactor )

definition ‹rear_Car = [[racing_car | (λ_. pos_speed ∩ on_lane1)]]rear_car›
definition ‹front_Car = [[racing_car | (λ_. pos_speed ∩ on_lane1)]]front_car›

term ‹dynamic_system = (rear_Car [[range time_lapse ]] front_Car [[range time_lapse ]] demon 0.5
A)›
```

Obviously, this is not necessarily deadlock free. But since we are only interested in traces, I think this as acceptable. Anyway, the driving strategy "K " will always produce deadlocks, for example.

3.3.5 Scenario Examples

```
definition disk z = {(x,y). x*x + y*y ≤ z}
term fun_upd
term I (i0::nat := x)



term
  [sync (|global_state = (mt(rear_car := (|pos       = (0.0,0.0)::real×real,
                                          speed     = (33.3,0.0),   — 120 km/h
                                          acc       = (1.0,0.0),
                                          acc_range = {(1.0,0.0), (0.0,1.0), (0.0,−1.0),
```

```
                                        (−10.0,0.0), (−20.0,0.0)},
                              extension_field = disk 4.0 |),
                    front_car  := (|pos       = (25.0,−3.0)::real×real,
                              speed     = (33.3,0.0),  — 120 km/h
                              acc       = (0.0,0.0),
                              acc_range = {(1.0,0.0),  (0.0,1.0), (0.0,−1.0),
                                        (−10.0,0.0), (−20.0,0.0)},
                              extension_field = disk 4.0 |)))
      |),
time_lapse 0.5, — 0.5 s later
sync (|global_state = (mt(rear_car  := (|pos       = (16.7,0.0)::real×real,
                              speed     = (33.3,0.0),   — 120 km/h
                              acc       = (0.0,0.0),
                              acc_range = {(10.0,0.0),  (2.0,3.0), (2.0,−3.0),
                                        (−10.0,0.0), (−20.0,0.0)},
                              extension_field = disk 4.0 |),
                    front_car  := (|pos       = (50.0,−2.0)::real×real,
                              speed     = (50.0,0.0),  — 120 km/h
                              acc       = (0.0,1.0),   — move right
                              acc_range = {(1.0,0.0),  (0.0,1.0), (0.0,−1.0),
                                        (−10.0,0.0), (−20.0,0.0)},
                              extension_field = disk 4.0 |)))
      |),
time_lapse 0.5, — 0.5 s later
sync (|global_state = (mt(rear_car  := (|pos       = (33.3,0.0)::real×real,
                              speed     = (33.3,0.0),   — 120 km/h
                              acc       = (0.0,0.0),
                              acc_range = {(1.0,0.0),  (0.0,1.0), (0.0,−1.0),
                                        (−10.0,0.0), (−20.0,0.0)},
                              extension_field = disk 4.0 |),
                    front_car  := (|pos       = (66.7,−2.0)::real×real,
                              speed     = (33.3,0.0),  — 120 km/h
                              acc       = (0.0,1.0),   — move right
                              acc_range = {(10.0,0.0),  (2.0,3.0), (2.0,−3.0),
                                        (−10.0,0.0), (−20.0,0.0)},
                              extension_field = disk 4.0 |)))
      |),
time_lapse 0.5, — 0.5 s later
sync (|global_state = (mt(rear_car  := (|pos       = (50.0,0.0)::real×real,
                              speed     = (33.3,0.0),   — 120 km/h
                              acc       = (0.0,0.0),
                              acc_range = {(10.0,0.0),  (2.0,3.0), (0.0,−3.0),
                                        (−10.0,0.0), (−20.0,0.0)},
                              extension_field = disk 4.0 |),
                    front_car  := (|pos       = (66.7,−2.0)::real×real,
                              speed     = (33.3,0.0),  — 120 km/h
                              acc       = (0.0,1.0),   — move right
                              acc_range = {(10.0,0.0),  (2.0,3.0), (2.0,−3.0),
                                        (−10.0,0.0), (−20.0,0.0)},
```

```
                                    extension_field = disk 4.0 |)))
      |)]

term‹map ev
    [sync (|global_state = (mt(rear_car  := (|pos        = (0.0,0.0)::real×real,
                                    speed     = (33.3,0.0),   — 120 km/h
                                    acc       = (1.0,0.0),
                                    acc_range = {(1.0,0.0), (0.0,1.0), (0.0,−1.0),
                                                 (−10.0,0.0), (−20.0,0.0)},
                                    extension_field = disk 4.0 |),
                             front_car  := (|pos       = (25.0,−3.0)::real×real,
                                    speed     = (33.3,0.0),  — 120 km/h
                                    acc       = (0.0,0.0),
                                    acc_range = {(1.0,0.0), (0.0,1.0), (0.0,−1.0),
                                                 (−10.0,0.0), (−20.0,0.0)},
                                    extension_field = disk 4.0 |)))
       |),
    time_lapse 0.5, — 0.5 s later
    sync (|global_state = (mt(rear_car  := (|pos       = (16.7,0.0)::real×real,
                                    speed     = (33.3,0.0),   — 120 km/h
                                    acc       = (0.0,0.0),
                                    acc_range = {(10.0,0.0), (2.0,3.0), (2.0,−3.0),
                                                 (−10.0,0.0), (−20.0,0.0)},
                                    extension_field = disk 4.0 |),
                             front_car  := (|pos       = (50.0,−2.0)::real×real,
                                    speed     = (50.0,0.0),  — 120 km/h
                                    acc       = (0.0,1.0),   — move right
                                    acc_range = {(1.0,0.0), (0.0,1.0), (0.0,−1.0),
                                                 (−10.0,0.0), (−20.0,0.0)},
                                    extension_field = disk 4.0 |)))
      |),
    time_lapse 0.5, — 0.5 s later
    sync (|global_state = (mt(rear_car  := (|pos       = (33.3,0.0)::real×real,
                                    speed     = (33.3,0.0),   — 120 km/h
                                    acc       = (0.0,0.0),
                                    acc_range = {(1.0,0.0), (0.0,1.0), (0.0,−1.0),
                                                 (−10.0,0.0), (−20.0,0.0)},
                                    extension_field = disk 4.0 |),
                             front_car  := (|pos       = (66.7,−2.0)::real×real,
                                    speed     = (33.3,0.0),  — 120 km/h
                                    acc       = (0.0,1.0),   — move right
                                    acc_range = {(10.0,0.0), (2.0,3.0), (2.0,−3.0),
                                                 (−10.0,0.0), (−20.0,0.0)},
                                    extension_field = disk 4.0 |)))
      |),
    time_lapse 0.5, — 0.5 s later
    sync (|global_state = (mt(rear_car  := (|pos       = (33.3,0.0)::real×real,
                                    speed     = (33.3,0.0),   — 120 km/h
                                    acc       = (0.0,0.0),
```

$$
\begin{aligned}
&\qquad\qquad acc\_range = \{(10.0,0.0),\ (2.0,3.0),\ (2.0,-3.0),\\
&\qquad\qquad\qquad (-10.0,0.0),\ (-20.0,0.0)\},\\
&\qquad\qquad extension\_field = disk\ 4.0\ \|),\\
&front\_car\ := (\|pos \quad = (66.7,-2.0) :: real \times real,\\
&\qquad\qquad speed \quad = (33.3,0.0), \text{ — 120 km/h}\\
&\qquad\qquad acc \quad = (0.0,1.0), \text{ — move right}\\
&\qquad\qquad acc\_range = \{(10.0,0.0),\ (2.0,3.0),\ (2.0,-3.0),\\
&\qquad\qquad\qquad (-10.0,0.0),\ (-20.0,0.0)\},\\
&\qquad\qquad extension\_field = disk\ 4.0\ \|)))\\
&\|)]
\end{aligned}
$$

$\in \mathcal{T}\ (demon\ 0.5\ Actors\ [\![S]\!]\ front\_car_P\ [\![S]\!]\ rear\_car_P)\rangle$

end

4 A Safety-Property in Autonomous Cars: RSS in a 2-cars-Scenario

theory *RSS_2cars*
 imports *CSP−AutoCars*
begin

The instance of the framework for a 2 cars scenario: One-lane topology, topology is identical to simple vector-space *real*, no backwards-driving. Simple demon.

This section discusses a 2-cars scenario where both cars respect the RSS strategy. We refer to the definitions for RSS in [9].

abbreviation $rear_{id}$::*nat* **where** $rear_{id} \equiv 0$
abbreviation $front_{id}$::*nat* **where** $front_{id} \equiv 1$

4.1 Global Parameters of the Scenario-Class

locale *RSS_2cars_samelane_samedirection* = — 2 cars, one lane, one direction

 fixes $a_{maxaccel}$ ::*real* **and** $a_{minbrake}$::*real* **and** $a_{maxbrake}$::*real* **and** ϱ::*real*
 fixes d_{real} :: ‹*real scene* ⇒ *real*› — signed distance between the two vehicles
  and d_{safe} :: ‹*real scene* ⇒ *real*› — safe distance between the two vehicles
 fixes rss_{motion}::‹*real* $motion_{acc}$› **and** $kinetics_{RSS}$:: ‹*real* $motion_{acc}$ ⇒ *real motion*›

 assumes *a0*:‹$0 < a_{maxaccel}$›
  and *a1*:‹$0 < a_{minbrake}$›
  and *a2*:‹$a_{minbrake} \leq a_{maxbrake}$›
  and *a3*:‹$\varrho > 0$›

— safe logitudinal distance as defined by RSS, in section 3.1
defines *d0*:‹$d_{safe}\ \sigma_g \equiv max\ 0\ (\varrho * speed\ (\sigma_g\ rear_{id}) + ((\varrho^2)/2) * a_{maxaccel} + (speed\ (\sigma_g\ rear_{id}) + \varrho * a_{maxaccel})^2\ /\ (2 * a_{minbrake}) - (speed\ (\sigma_g\ front_{id}))^2\ /\ (2 * a_{maxbrake}))$›
 — To improve: use current *acc* $(\sigma_g\ rear_{id})$ is more precise than $a_{maxaccel}$
 and *d1*:‹$d_{real}\ \sigma_g \equiv pos(\sigma_g\ front_{id}) - pos(\sigma_g\ rear_{id})$›
 — signed distance between the two vehicles

 assumes *r0*:‹$rss_{motion}\ 0\ \delta t\ \sigma_g \equiv \{..\ if\ d_{safe}\ \sigma_g > d_{real}\ \sigma_g\ then\ -a_{minbrake}\ else\ a_{maxaccel}\}$›
  — possible accelerations for v_0 or c_r (rear car)
  — le pb est qu'ici on ne tient pas compte du délai de réaction : la vitesse connue au moment de la décision d'ego est la vitesse à $t-\varrho$??
  and *r1*:‹$rss_{motion}\ 1\ \delta t\ \sigma_g \equiv \{-a_{maxbrake}\ ..\}$›
  — possible accelerations for v_1 or c_f (front car)

— RSS kinetics are not backward
defines $k0$:⟨$kinetics_{RSS}$ $motion_{acc}$ ≡
λ i δt σ_g. { if speed $(\sigma_g\ i) + (\delta t * a) > 0$
then (| pos = pos $(\sigma_g\ i) + (\delta t) * speed\ (\sigma_g\ i) + ((\delta t\hat{}2)/2) * a$,
speed = speed $(\sigma_g\ i) + (\delta t * a)$,
acc = a |)
else (| pos = pos $(\sigma_g\ i) - (speed\ (\sigma_g\ i))^2\ /\ (2 * a)$,
speed = 0,
acc = 0 |)
— the $-(speed\ (\sigma_g\ i))^2\ /\ (2 * a)$ comes from the newtonian kinetics formula with $t_1{=}{-}v/a$, i.e. the time when $v = v_0 + a * t_1 = 0$ (the vehicle stops at $t_1{\in}[t,\ t{+}\delta t]$)
| a. a ∈ $motion_{acc}$ i δt σ_g }⟩

begin

4.2 Relations between Driving Strategies

lemma $kinetics_{RSS}$_empty: ⟨$kinetics_{RSS}$ $motion_{acc}$ i δt σ_g = {} ⟷ $motion_{acc}$ i δt σ_g = {}⟩
by (auto simp add: k0)

difference between the min braking distance of c_f and the max braking distance of c_r

definition ⟨$d_{min} \equiv \lambda\ (\sigma_g$::real scene). max 0 $((speed\ (\sigma_g\ rear_{id}))^2\ /\ (2 * a_{minbrake}) - (speed\ (\sigma_g\ front_{id}))^2\ /\ (2 * a_{maxbrake}))$⟩

declare minus_divide_le_eq[simp] not_less[simp]
theorem rss_is_safe_2cars:
assumes a: ⟨$d_{min}\ \sigma_{g0} < d_{real}\ \sigma_{g0} \wedge 0 \leq speed\ (\sigma_{g0}\ 0) \wedge 0 \leq speed\ (\sigma_{g0}\ 1)$⟩
— pquoi une inégalité stricte?? Est-ce par définition de is_{safe}
and b: ⟨$0 < \Delta t \wedge \Delta t < \varrho$⟩ — et ici, pourquoi inégalité stricte??
shows ⟨is_{safe} {0, 1} ($kinetics_{RSS}$ rss_{motion}) Δt σ_{g0} 0⟩
proof –
define P_{inv} **where** ⟨$P_{inv} \equiv \lambda\ (\sigma_g$::real scene). $0 \leq speed\ (\sigma_g\ rear_{id}) \wedge 0 \leq speed\ (\sigma_g\ front_{id}) \wedge d_{min}\ \sigma_g < d_{real}\ \sigma_g$⟩
— invariant: the distance between the two cars is d_{min}
show ?thesis
proof(rule is_{safe}_rule_inv[**where** P_{inv}=⟨$\lambda\ \sigma_g$ sid. $P_{inv}\ \sigma_g$⟩])
— use is_{safe}_rule_inv with P_{inv} defined above
show ⟨$0 < \Delta t$⟩ **using** b **by** simp
next
show ⟨$P_{inv}\ \sigma_{g0}$⟩
using a **by** (auto simp add:P_{inv}_def d0)
next
fix σ_g δt
show ⟨$P_{inv}\ \sigma_g \wedge \delta t \in \{0{<}..\Delta t\} \Longrightarrow \forall\, i{\in}\{0,\ 1\}.\ kinetics_{RSS}\ rss_{motion}\ i\ \delta t\ \sigma_g \neq \{\}$⟩
using a0 a1 a2 r0 r1 P_{inv}_def **by** (auto simp:$kinetics_{RSS}$_empty)
next

```
  fix σ_g δt
  show ‹P_inv σ_g ∧ δt ∈ {0<..Δt} ⟹ ∀σ_g'∈{σ_g'. ∀i∈{0, 1}. σ_g' i ∈ kinetics_RSS rss_motion i
δt σ_g}. P_inv σ_g'›
  proof(auto simp:P_inv_def)
    fix σ_g'
    assume ‹σ_g' (0::nat) ∈ kinetics_RSS rss_motion 0 δt σ_g› and ‹0 < δt›
    then show ‹0 ≤ speed (σ_g' rear_id)›
      by(auto simp:k0 r0 mult.commute)
  next
    fix σ_g'
    assume ‹σ_g' (1::nat) ∈ kinetics_RSS rss_motion 1 δt σ_g› and ‹0 < δt›
    then show ‹0 ≤ speed (σ_g' front_id)›
      by(auto simp:k0 r1 mult.commute)
  next
    fix σ_g'
    assume as1:‹0 ≤ speed (σ_g rear_id)›
       and as2:‹0 ≤ speed (σ_g front_id)›
       and as3:‹d_min σ_g < d_real σ_g›
       and as4:‹0 < δt›
       and as5:‹δt ≤ Δt›
       and as6:‹σ_g' (0::nat) ∈ kinetics_RSS rss_motion 0 δt σ_g›
       and as7:‹σ_g' front_id ∈ kinetics_RSS rss_motion 1 δt σ_g›

    let ?a_0 = ‹acc(σ_g' rear_id)› and ?a_1 = ‹acc(σ_g' front_id)›
       and ?b_0 = ‹(speed (σ_g rear_id))^2 / (2*a_minbrake)› and ?b_1 = ‹(speed (σ_g front_id))^2 /
(2*a_maxbrake)›
      and ?b'_0 = ‹(speed (σ_g' rear_id))^2 / (2*a_minbrake)› and ?b'_1 = ‹(speed (σ_g' front_id))^2 /
(2*a_maxbrake)›


    — some simplifications factorized here
    have simp0:‹0 ≤ speed (σ_g' rear_id)› and simp1:‹0 ≤ speed (σ_g' front_id)›
      using as6 as7 by (auto simp:k0 r0 r1 mult.commute)

     have simp2:‹speed (σ_g rear_id) / 2 + ?a_0*δt / 2 ≥ 0› and simp3:‹speed (σ_g front_id) / 2 +
?a_1*δt / 2 ≥ 0›
      using as6 as7 by (auto simp add:k0 r0 r1 as4 as2 as1 mult.commute)

    have simp4:‹speed (σ_g rear_id) + ?a_0*δt / 2 ≥ 0› and simp5:‹speed (σ_g front_id) + ?a_1*δt / 2
≥ 0›
      using as1 simp2 apply linarith
      using as2 simp3 by linarith

     have simp6:‹δt*speed (σ_g rear_id) + ?a_0*δt^2 / 2 ≥ 0› and simp7:‹δt*speed (σ_g front_id) +
?a_1*δt^2 / 2 ≥ 0›
      using as4 simp4 simp5
      by (metis (no_types, hide_lams) distrib_left less_eq_real_def
                          mult.assoc mult.commute mult_nonneg_nonneg power2_eq_square
times_divide_eq_right)+
```

```
have simp4p:‹speed (σg rearid) + ?a0*δt / 2 ≥ speed (σg rearid) / 2›
 and simp5p:‹speed (σg frontid) + ?a1*δt / 2 ≥ speed (σg frontid) / 2›
  using as1 simp2 apply linarith
  using as2 simp3 by linarith

have simp6p:‹δt*speed (σg rearid) + ?a0*δt^2 / 2 ≥ δt*speed (σg rearid) / 2›
 and simp7p:‹δt*speed (σg frontid) + ?a1*δt^2 / 2 ≥ δt*speed (σg frontid) / 2›
  using as4 simp4p simp5p
  by (metis distrib_left less_eq_real_def mult.left_commute mult_left_mono
           power2_eq_square times_divide_eq_right)+

have simp8:‹speed (σg rearid) + a * δt ≤ 0 ⟹ − (speed (σg rearid))^2 / (a * 2) ≤ (δt*speed
(σg rearid)) / 2› for a
  using as1 mult_left_mono[where b=‹δt› and a=‹−speed (σg rearid) / a› and c=‹speed (σg
rearid)›, OF _ as1]
  apply (auto simp add:mult.commute power2_eq_square split:if_splits)
  using as4 mult_pos_pos by force

have simp9:‹speed (σg frontid) + a * δt ≤ 0 ⟹ − (speed (σg frontid))^2 / (a * 2) ≤ (δt*speed
(σg frontid)) / 2› for a
  using as2 mult_left_mono[where b=‹δt› and a=‹−speed (σg frontid) / a› and c=‹speed (σg
frontid)›, OF _ as2]
  apply (auto simp add:mult.commute power2_eq_square split:if_splits)
  using as4 mult_pos_pos by force

— safety distance proofs
{ assume ‹speed (σg' frontid) = 0›
  then have ‹max 0 ?b1 ≤ pos (σg' frontid) − pos (σg frontid)›
   using as7 as2 as4 apply (auto simp add:k0 r1 mult.commute divide_le_0_iff split:if_splits)

    apply (smt (z3) zero_less_mult_iff)
    using divide_left_mono[where c=‹(speed (σg frontid))^2› and a=‹(amaxbrake * 2)›]
    by (smt (z3) divide_cancel_left minus_divide_right mult_minus_left
           zero_eq_power2 zero_le_mult_iff zero_le_power)
} note dmin_front_car_stopping=this

{  assume h0:‹speed (σg' frontid) > 0›
  from as7 have h1:‹− ?a1 / amaxbrake ≤ 1›
    by (auto simp add:k0 r1) (smt (z3) a1 a2 divide_le_eq_1_neg minus_divide_right)
  have h2:‹(speed (σg frontid) + δt * ?a1)^2 / (2*amaxbrake)
          =  (speed (σg frontid))^2 / (2*amaxbrake) + (?a1/amaxbrake) * (δt * speed (σg
frontid) + ?a1 * δt^2 / 2)›
    by (simp add: power2_sum distrib_left mult.commute power2_eq_square
             add_divide_distrib mult.left_commute)
  have ‹0 ≤ pos (σg' frontid) − pos (σg frontid)›
    using as7 h0 simp7 by (auto simp add:k0 r1 mult.commute split:if_splits)
  then have ‹max 0 (?b1 − ?b'1) ≤ pos (σg' frontid) − pos (σg frontid)›
    using as7 h0 mult_left_mono[OF h1 simp7] h2 a1 a2
```

```
      by (auto simp add:k0 r1 mult.commute split:if_splits)
  } note dmin_front_car_motion=this

  have front_car:‹max 0 (?b_1 − ?b'_1) ≤ pos (σ_g' front_id) − pos (σ_g front_id)›
    using dmin_front_car_motion dmin_front_car_stopping simp1 by fastforce

  { assume h0:‹d_safe σ_g > d_real σ_g› and h1:‹speed (σ_g' rear_id) > 0›
    have h2:‹?a_0 / a_minbrake ≤ −1›
      using as6 a1 a2 h0 h1 apply(auto simp add:k0 r0 split:if_splits)
      by (metis a1 add.inverse_inverse le_divide_eq_1_pos minus_divide_left neg_le_iff_le)
    have ‹pos (σ_g' rear_id) − pos (σ_g rear_id) ≤ (speed (σ_g rear_id))^2 / (2∗a_minbrake) − (speed
(σ_g' rear_id))^2 / (2∗a_minbrake)›
      using as6 a1 a2 h0 h1 mult_left_mono[OF h2 simp6]
          by (auto simp add:k0 r0 power2_sum add_divide_distrib distrib_left mult.commute
split:if_splits)
         (smt (verit) mult.assoc mult.commute power2_eq_square)
    then have ‹?b'_0 − ?b_0 ≤ pos (σ_g rear_id) − pos (σ_g' rear_id)›
      by force
  } note dmin_rear_car_nonsafe_breaking=this

  { assume h0:‹d_safe σ_g > d_real σ_g› and h1:‹speed (σ_g' rear_id) = 0›
    have h2:‹a ≤ −a_minbrake ⟹ a / a_minbrake ≤ −1› for a
      by (metis a1 divide_eq_minus_1_iff divide_le_cancel less_le not_le)
    have h4:‹pos (σ_g' rear_id) − pos (σ_g rear_id) ≤ (speed (σ_g rear_id))^2 / (2∗a_minbrake)›
         using as6 a1 a2 h0 h1 apply(auto simp add:k0 r0 power2_sum add_divide_distrib
split:if_splits)
      using mult_left_mono[OF h2] by (simp add: mult.commute)
    then have ‹− ?b_0 ≤ pos (σ_g rear_id) − pos (σ_g' rear_id)›
      by argo
  } note rear_car_nonsafe_stopping=this

  have dmin_rear_car_nonsafe:‹d_safe σ_g > d_real σ_g ⟹ ?b'_0 − ?b_0 ≤ pos (σ_g rear_id) − pos
(σ_g' rear_id)›
    using dmin_rear_car_nonsafe_breaking rear_car_nonsafe_stopping simp0 by fastforce

  { assume h0:‹d_safe σ_g ≤ d_real σ_g› and h1:‹speed (σ_g' rear_id) = 0›
    have h2: ‹(δt ∗ speed (σ_g rear_id)) / 2 ≤ ϱ∗speed(σ_g rear_id)›
      using as1 as5 b less_eq_real_def by force
    have h3:‹pos (σ_g' rear_id) − pos (σ_g rear_id) ≤ ϱ∗speed(σ_g rear_id)›
       using as6 as1 a1 h0 h1 apply(auto simp add:d0 d_min_def k0 r0 mult.commute not_le
split:if_splits)
      using order_trans[OF simp8 h2]
      apply (smt (verit, best) mult.commute neg_divide_le_eq)
      by (metis add_cancel_right_right add_le_less_mono as4 mult_eq_0_iff
              mult_less_cancel_left_pos not_less)
    have h4: ‹?b_0 ≤ (speed(σ_g rear_id) + ϱ∗a_maxaccel)^2/(2∗a_minbrake)›
     by (smt (verit, ccfv_SIG) a0 a1 a3 as1 frac_le mult_pos_pos power_mono zero_le_power2)
    with h0 have h5:‹ϱ∗speed(σ_g rear_id) + ?b_0 − ?b_1 < pos (σ_g front_id) − pos (σ_g rear_id)›
      by (smt (verit, best) a0 a3 d0 d1 divide_pos_pos zero_less_mult_iff zero_less_power)
```

```
    with h3 have ‹?b0 − ?b1 < pos (σg frontid) − pos (σg' rearid)›
      by fastforce
  } note dmin_rear_car_safe_stopping=this

  { assume h0:‹dsafe σg ≤ dreal σg› and h1:‹speed (σg' rearid) > 0›
    have h2:‹δt∗speed(σg rearid) ≤ ϱ∗speed(σg rearid)›
      using as1 as4 as5 b mult_right_less_imp_less by force
    from as6 have h3:‹?a0∗δt²/2 < amaxaccel∗ϱ²/2›
      using a0 a1 a3 h0 apply (auto simp add:k0 r0 as4 split:if_splits)
      by (smt (verit) as4 as5 b eq_divide_imp frac_le mult.commute power2_le_imp_le
              zero_eq_power2 zero_le_mult_iff)
    from as6 have h4: ‹?b'0 ≤ (speed(σg rearid) + ϱ∗amaxaccel)²/(2∗aminbrake)›
      using a1 h0 apply (auto simp add:k0 r0)
         by (smt (verit, best) a0 a1 as4 as5 b divide_le_cancel mult_mono' power_mono
zero_le_mult_iff)
    have h5: ‹?b0 ≤ (speed(σg rearid) + ϱ∗amaxaccel)²/(2∗aminbrake)›
     by (smt (verit, ccfv_SIG) a0 a1 a3 as1 frac_le mult_pos_pos power_mono zero_le_power2)
    from h2 h3 h4 h5
    have ‹ δt∗speed(σg rearid) + ?a0∗δt²/2 + (max ?b0 ?b'0)
                    < ϱ∗speed(σg rearid) + amaxaccel∗ϱ²/2 + (speed(σg rearid) +
ϱ∗amaxaccel)²/(2∗aminbrake)›
      by (smt (verit) mult.commute)
     with h0 have h6:‹δt∗speed(σg rearid) + ?a0∗δt²/2 + (max ?b0 ?b'0) − ?b1 < pos (σg
frontid) − pos (σg rearid) ›
      by (smt (verit, ccfv_SIG) d0 d1 mult.commute times_divide_eq_right)
    have ‹pos (σg' rearid) − pos (σg rearid) = δt∗speed(σg rearid) + ?a0∗δt²/2›
   using as6 as1 a1 h1 by (auto simp add:d0 dmin_def k0 r0 mult.commute not_le split:if_splits)

    with h6 have ‹(max ?b0 ?b'0) − ?b1 < pos (σg frontid) − pos (σg' rearid)›
      by fastforce
  } note dmin_rear_car_safe_motion=this

  have dmin_rear_car_safe:‹dsafe σg ≤ dreal σg ⟹ (max ?b0 ?b'0) − ?b1 < pos (σg frontid)
− pos (σg' rearid)›
    by (metis a1 divide_le_cancel less_eq_real_def max_def mult_2 power2_less_eq_zero_iff
           power_zero_numeral dmin_rear_car_safe_motion dmin_rear_car_safe_stopping
simp0
          zero_less_double_add_iff_zero_less_single_add)

  { assume h0:‹dsafe σg ≤ dreal σg›
    then have ‹0 ≤ pos (σg' rearid) − pos (σg rearid)›
      using h0 as1 as6 simp6 as4 apply (auto simp add:k0 r0 mult.commute)
      by (metis add.right_neutral add_mono_thms_linordered_field(4) divide_nonneg_nonpos
             not_le not_numeral_less_zero zero_le_power2 zero_less_mult_iff)
  } note dmin_rear_car_forward=this

  have dmin:‹?b'0 − ?b'1 < dreal σg'›
    using as3 d1 dmin_def front_car dmin_rear_car_nonsafe dmin_rear_car_safe by fastforce
```

— no collision's proofs

```
{ assume h0:‹?b₀ ≤ ?b₁› and h1:‹?b'₀ < ?b'₁›
  — ?b'0 < ?b'1, is added to profit from previous results of dmin
    and h2:‹speed (σ_g' rear_id) = 0›
  from h0 have h3:‹speed (σ_g rear_id) ≤ speed (σ_g front_id)›
    using mult_right_mono[OF h0, where c=‹(2 * a_minbrake)›, simplified]
   by (smt (z3) a1 a2 as2 less_divide_eq mult_left_mono power2_le_imp_le zero_le_power2)

  from h2 have h6:‹pos (σ_g' rear_id) − pos (σ_g rear_id) ≤ δt * speed (σ_g rear_id) / 2›
    using as6 as1 apply(auto simp add:k0 r0 a1)
    using simp8[simplified divide_divide_eq_left[symmetric]]
     apply (smt (verit, ccfv_threshold) mult.assoc mult.left_commute
        mult_left_mono mult_minus_left neg_le_iff_le power2_eq_square real_add_le_0_iff)
    by (metis add_cancel_left_right add_le_less_mono as4 mult_eq_0_iff
            mult_less_cancel_left_pos not_less)
  have h6:‹pos (σ_g' rear_id) − pos (σ_g rear_id) ≤ pos (σ_g' front_id) − pos (σ_g front_id)›
    using as7 h1 h2 apply(auto simp add:k0 r1 a1)
    by (smt (verit, best) as.select_convs(3) as4 field_sum_of_halves h3 h6
                   mult.commute mult_less_cancel_left_pos simp7p)
} note no_collision_if_slow_rear_car_stopping=this

{ assume h0:‹?b₀ ≤ ?b₁› and h1:‹?b'₀ < ?b'₁›
    and h2:‹speed (σ_g' rear_id) > 0›
  from h0 have h3:‹speed (σ_g rear_id) ≤ speed (σ_g front_id)›
    using mult_right_mono[OF h0, where c=‹(2 * a_minbrake)›, simplified]
   by (smt (z3) a1 a2 as2 less_divide_eq mult_left_mono power2_le_imp_le zero_le_power2)

  have h4:‹speed (σ_g' front_id) > 0›
    by (metis a1 div_0 divide_nonneg_pos h1 mult.commute mult_2_right not_less
            order.not_eq_order_implies_strict simp1 zero_le_power2
            zero_less_double_add_iff_zero_less_single_add zero_power2)
  from h1 h2 h3 h4 have h5:‹speed (σ_g rear_id) + ?a₀ * δt < speed (σ_g front_id) + ?a₁ * δt›
    using as6 as7 apply(auto simp add:k0 r0 r1)
    by (smt (verit, best) a1 a2 frac_le mult.commute power_mono zero_le_power2)
  have h6:‹δt * speed (σ_g rear_id) + δt² * ?a₀ < δt * speed (σ_g front_id) + δt² * ?a₁›
    by (metis (no_types, hide_lams) as4 distrib_left h5 mult.commute mult.left_commute
            mult_less_cancel_left_pos power2_eq_square)
  with h3 have h7:‹δt * speed (σ_g rear_id) + δt² * ?a₀ / 2 < δt * speed (σ_g front_id) + δt² *
?a₁ / 2›
    by (smt (verit, ccfv_SIG) as4 field_sum_of_halves mult_less_cancel_left_pos)
  have ‹pos (σ_g' rear_id) − pos (σ_g rear_id) < pos (σ_g' front_id) − pos (σ_g front_id)›
    using as6 as7 h2 h4 h7 by (auto simp add:k0 r0 r1 a1 d_min_def d1
                                   divide_nonneg_pos power2_sum not_le)
} note no_collision_if_slow_rear_car_nonstopping=this
```

remaining case (REM):

1. non-safe mode *dsafe* > *dreal*: in the opposite case, we profit from previous results of *dmin_rear_car_safe*

2. faster rear car : but it becomes slower after breaking

{ **assume** $h0$:‹$?b_1 < ?b_0$› **and** $h1$:‹$d_{safe}\ \sigma_g > d_{real}\ \sigma_g$› **and** $h2$:‹$?b'_0 < ?b'_1$›
  and $h3$:‹speed $(\sigma_g{}'\ front_{id}) = 0$›
 then have False
  by (metis a1 divide_eq_0_iff le_divide_eq mult_eq_0_iff not_less power_zero_numeral
   zero_le_power2 zero_less_numeral)
} **note** REM_front_car_not_stopping=this

{ **assume** $h0$:‹$?b_1 < ?b_0$› **and** $h1$:‹$d_{safe}\ \sigma_g > d_{real}\ \sigma_g$› **and** $h2$:‹speed $(\sigma_g{}'\ front_{id}) > 0$›
  and $h3$:‹speed$(\sigma_g\ front_{id}) - a_{maxbrake} * \delta t \leq 0$›
 have $h4$:‹$?b_1 \leq$ pos $(\sigma_g{}'\ front_{id})$ − pos $(\sigma_g\ front_{id})$›
  using as7 h2 **apply** (auto simp add:k0 r1 mult.commute not_le power2_sum
   add_divide_distrib split:if_splits)
  by (smt (z3) as.select_convs(3) divide_minus_left h3 minus_divide_right
   mult_minus_left simp7p simp9)
 then have ‹$0 <$ pos $(\sigma_g{}'\ front_{id})$ − pos $(\sigma_g{}'\ rear_{id})$›
  by (smt (verit, best) a1 as3 d1 d_{min}_def divide_nonneg_pos
   dmin_rear_car_nonsafe h1 zero_le_power2)
} **note** REM_front_car_may_stop=this

{ **assume** $h0$:‹$?b_1 < ?b_0$› **and** $h1$:‹$d_{safe}\ \sigma_g > d_{real}\ \sigma_g$› **and** $h2$:‹speed $(\sigma_g{}'\ front_{id}) > 0$›
  and $h3$:‹speed$(\sigma_g\ front_{id}) - a_{maxbrake} * \delta t > 0$›

 have $s1$:‹$a_{maxbrake} / a_{minbrake} \geq 1 \wedge$ sqrt $a_{maxbrake} \geq$ sqrt $a_{minbrake} \wedge$ sqrt $a_{minbrake} > 0$›
  using a1 a2 **by** simp
 then have $s2$:‹$a_{maxbrake} / a_{minbrake} \geq$ sqrt $a_{maxbrake}$ / sqrt $a_{minbrake}$›
  by (smt (verit, ccfv_threshold) divide_less_eq_1 real_div_sqrt real_sqrt_divide)
 then have $s3$:‹$a_{maxbrake}$ * sqrt $a_{minbrake} \geq a_{minbrake}$ * sqrt $a_{maxbrake}$›
  by (smt (verit, ccfv_SIG) a1 a2 divide_divide_times_eq divide_le_eq_1_pos
   divide_less_eq_1 mult_nonneg_nonneg real_sqrt_gt_0_iff)

 have $s4$:‹sqrt $a_{maxbrake}$ / sqrt $a_{minbrake} \geq$ (speed $(\sigma_g\ front_{id})$) / (speed $(\sigma_g\ rear_{id})$)›
  using real_sqrt_le_mono[OF order.strict_implies_order[OF h0]]
  apply(simp add:real_sqrt_divide real_sqrt_mult as1 as2 mult.commute)
  apply(subst (asm) (1 2) divide_divide_eq_left[symmetric], subst(asm) divide_le_cancel, simp)
  by (smt (verit, ccfv_threshold) as1 divide_less_eq_1 mult.commute pos_divide_le_eq
   s1 times_divide_eq_left)
 have $s5$:‹speed$(\sigma_g\ rear_{id})$ / $a_{minbrake} \geq$ speed$(\sigma_g\ front_{id})$ / $a_{maxbrake}$›
  using order_trans[OF s4 s2] **apply**(subst (asm) pos_divide_le_eq)
  apply (smt (verit, ccfv_threshold) a1 a2 as1 divide_eq_0_iff divide_nonneg_pos h0
   zero_le_power2 zero_power2)
  apply (simp add: a1 le_divide_eq mult.commute)
  by (smt (verit) a1 a2 divide_le_cancel nonzero_mult_div_cancel_left)
 with h3 **have** $s6$:‹speed$(\sigma_g\ rear_{id}) - a_{minbrake} * \delta t > 0$›
  using h3 a1 a2 **by** (metis a1 diff_gt_0_iff_gt less_le_trans mult.commute pos_less_divide_eq)

```
  have s7:‹(speed (σ_g front_id)) * sqrt a_minbrake ≤ (speed (σ_g rear_id)) * sqrt a_maxbrake›
    using real_sqrt_le_mono[OF order.strict_implies_order[OF h0]]
    apply(simp add:real_sqrt_divide real_sqrt_mult as1 as2 mult.commute)
     apply(subst (asm) (1 2) divide_divide_eq_left[symmetric], subst(asm) divide_le_cancel,
simp)
    apply(subst (asm) pos_divide_le_eq)
    using a1 a2 apply auto[1]
    by (simp add: a1 le_divide_eq mult.commute)
  have s8:‹  (speed (σ_g front_id) − a_maxbrake*δt) / sqrt a_maxbrake
         ≤ (speed (σ_g rear_id) − a_minbrake *δt) / sqrt a_minbrake›
    apply(subst pos_divide_le_eq) using a1 a2 apply auto
    apply(subst pos_le_divide_eq) apply (simp_all add:left_diff_distrib)
          using s7 s3 by (metis as4 diff_mono mult.commute mult.left_commute
mult_le_cancel_left_pos)
  with h3 have s9:‹(speed(σ_g front_id) − a_maxbrake *δt)^2/(2*a_maxbrake)
              ≤ ((speed(σ_g rear_id) − a_minbrake *δt)^2/(2*a_minbrake))›
  using abs_le_square_iff[where x = ‹(speed (σ_g front_id) − a_maxbrake*δt) / sqrt a_maxbrake›

                      and y = ‹(speed (σ_g rear_id) − a_minbrake *δt) / sqrt a_minbrake›]
    using s1 apply (auto simp add:power_divide)
    by (smt (verit, best) divide_less_0_iff s1)

  have s10:‹δt * speed (σ_g rear_id) / 2 ≤ (δt * speed (σ_g rear_id) − a_minbrake * δt^2 / 2)›
    by (smt (verit, best) as4 field_sum_of_halves le_divide_eq power2_eq_square
                 real_divide_square_eq s6 zero_less_power)

  have h4:‹δt*speed(σ_g front_id) − a_maxbrake *δt^2/2 ≤ pos (σ_g' front_id) − pos (σ_g front_id)›
    using as7 h2
    apply (auto simp add:k0 r1 mult.commute not_le power2_sum add_divide_distrib
          split:if_splits)
         by (metis comm_semiring_class.distrib mult_nonneg_nonneg real_0_le_add_iff
zero_le_power2)
  have ‹δt*speed(σ_g front_id) − a_maxbrake *δt^2/2 −?b_1 = −(speed(σ_g front_id) − a_maxbrake
*δt)^2/(2*a_maxbrake)›
    apply(simp add:power2_diff add_divide_distrib minus_add diff_add_eq_diff_diff_swap
         diff_divide_distrib)
    by (metis a1 a2 mult.assoc mult.left_commute nonzero_mult_div_cancel_left not_less
         power2_eq_square)
   with h4 have h5:‹pos (σ_g' front_id) − pos (σ_g front_id) − ?b_1 ≥ − (speed(σ_g front_id) −
a_maxbrake *δt)^2/(2*a_maxbrake)›
    by linarith

  have ‹a ≤ − a_minbrake ⟹  (δt * speed (σ_g rear_id) + a * δt^2 / 2)
                 ≤ (δt * speed (σ_g rear_id) − a_minbrake * δt^2 / 2)› for a
    by (smt (verit) as4 eq_divide_imp field_sum_of_halves mult_minus_left
           pos_minus_divide_le_eq zero_less_power2)
    then have h6:‹pos (σ_g' rear_id) − pos (σ_g rear_id) ≤  δt*speed(σ_g rear_id) − a_minbrake
*δt^2/2›
```

```
        using as6 h1 h2 a1
         apply (auto simp add:k0 r0 d0 d1 mult.commute not_le power2_sum add_divide_distrib
min_def
                    split:if_splits)
        using simp8 s10
        by (smt (verit, ccfv_SIG) divide_le_cancel eq_divide_imp mult.commute)

     have h7: ‹δt∗speed(σ_g rear_id) − a_minbrake ∗δt^2/2 − ?b_0 = −(speed(σ_g rear_id) − a_minbrake
∗δt)^2/(2∗a_minbrake)›
        apply(simp add:power2_diff add_divide_distrib minus_add diff_add_eq_diff_diff_swap
                      diff_divide_distrib)
       by (metis a1 less_eq_real_def mult.assoc mult.left_commute nonzero_mult_div_cancel_left

                  not_less power2_eq_square)

       with h5 h6 s9 have ‹pos (σ_g' rear_id) − pos (σ_g rear_id)  − ?b_0 ≤ pos (σ_g' front_id) − pos
(σ_g front_id) − ?b_1›
        by linarith
       then have ‹0 < pos (σ_g' front_id) − pos (σ_g' rear_id)›
        using as3 d1 d_min_def by auto
     } note REM_front_car_nonstopping=this

     have nocollision:‹0 < d_real σ_g'›
       apply (cases ‹?b'_0 ≥ ?b'_1›)
       using dmin apply simp
       apply(cases ‹?b_0 ≤ ?b_1›)
       using as3 d1 d_min_def no_collision_if_slow_rear_car_nonstopping
               no_collision_if_slow_rear_car_stopping simp0
        apply fastforce
       apply(cases ‹d_safe σ_g > d_real σ_g›, simp add:not_le)
                          using    REM_front_car_may_stop    REM_front_car_not_stopping    d1
REM_front_car_nonstopping simp1
        apply fastforce
       using d1 front_car dmin_rear_car_safe by force

     show ‹d_min σ_g' < d_real σ_g'›
       using dmin nocollision by (simp add: d_min_def)
   qed
  next
   fix σ_g
   show ‹P_inv σ_g ⟹ no_collision σ_g {0, 1} 0›
     unfolding d_min_def d1 P_inv_def no_collision_def by fastforce
  qed
qed


end

end
```

5 A Safety-Property in Autonomopus Cars: RSS in a N-cars-Scenario

theory RSS_Ncars
imports $CSP{-}AutoCars$
begin

Instantiation of the Framework for Single-Lane scenarios with N-cars, that alltogether use RSS.

declare $minus\_divide\_le\_eq[simp]$ $not\_less[simp]$

5.1 Global Parameters of the Scenario-Class

locale $RSS\_Ncars\_samelane\_same\_direction$ =

fixes $a_{maxaccel}$::$real$ **and** $a_{minbrake}$::$real$ **and** $a_{maxbrake}$::$real$ **and** ϱ::$real$
fixes d_{real}:: ‹$real\ scene \Rightarrow nat \Rightarrow real$› — actual distance between a vehicle and its front vehicle
and d_{rss}:: ‹$real\ scene \Rightarrow nat \Rightarrow real$› — safe longitudinal distance as defined by RSS
and d_{min}:: ‹$real\ scene \Rightarrow nat \Rightarrow real$› — safe longitudinal distance as defined by us ("RSS++")
fixes rss_{motion}::‹$real\ motion_{acc}$›

assumes $a0$:‹$0 < a_{maxaccel}$›
and $a1$:‹$0 < a_{minbrake}$›
and $a2$:‹$a_{minbrake} \leq a_{maxbrake}$›
and $a3$:‹$\varrho > 0$›

defines ‹$d_{min}\ \omega\ i \equiv max\ 0\ ((speed\ (\omega\ i))^2\ /\ (2 * a_{minbrake}) - (speed\ (\omega\ (i{+}1)))^2\ /\ (2 * a_{maxbrake}))$›
and $d0$:‹$d_{rss}\ \omega\ i \equiv max\ 0\ (\varrho * speed\ (\omega\ i) + ((\varrho^2)/2) * a_{maxaccel} + (speed\ (\omega\ i) + \varrho * a_{maxaccel})^2\ /\ (2 * a_{minbrake}) - (speed\ (\omega\ (i{+}1)))^2\ /\ (2 * a_{maxbrake}))$›
and $d1$:‹$d_{real}\ \omega\ i \equiv pos(\omega\ (i{+}1)) - pos(\omega\ i)$›

defines $r0$:‹$rss_{motion}\ i\ \delta t\ \omega \equiv if\ d_{rss}\ \omega\ i \leq d_{real}\ \omega\ i$
$then\ \{-a_{maxbrake}\ ..\ a_{maxaccel}\}$
$else\ \{-a_{maxbrake}\ ..\ -a_{minbrake}\}$›

begin

5.2 Relations between Driving Strategies

— RSS kinetics are not backward
definition $k0$:‹$kinetics_{RSS}\ motion_{acc} \equiv$

```
              λ i δt ω. { if speed (ω i) + (δt ∗ a) > 0
                    then (| pos = pos (ω i) + (δt) ∗ speed (ω i) + ((δt^2)/2) ∗ a,
                           speed = speed (ω i) + (δt ∗ a),
                           acc = a |)
                   else (| pos = pos (ω i) − (speed (ω i))^2 / (2 ∗ a), speed = 0, acc = 0 |)
                   |a. a ∈ motion_acc i δt ω }›

lemma kinetics_RSS_empty: ‹kinetics_RSS motion_acc i δt ω = {} ⟷ motion_acc i δt ω = {}›
  by (auto simp add: k0)

theorem rss_is_safe_N_cars:
    fixes N::nat
  assumes a: ‹∀ i∈{0..N}. d_min ω_0 i < d_real ω_0 i ∧ 0 ≤ speed (ω_0 i)›
      and b: ‹0 < Δt ∧ Δt < ϱ›
    shows ‹is_safe {0..N} (kinetics_RSS rss_motion) Δt ω_0 0›
proof −
  define P_inv where ‹P_inv ≡ λ (ω::real scene). (∀ i ≤ N. 0 ≤ speed (ω i)) ∧ (∀ i < N. d_min ω i
< d_real ω i)›

  show ?thesis
  proof(rule is_safe_rule_inv[where P_inv=‹λ ω sid. P_inv ω›])
    show ‹0 < Δt› using b by simp
  next
    show ‹P_inv ω_0›
      using a by (auto simp add:P_inv_def d0)
  next
    fix ω δt
    show ‹P_inv ω ∧ δt ∈ {0<..Δt} ⟹ ∀ i∈{0..N}. kinetics_RSS rss_motion i δt ω ≠ {}›
      using a0 a1 a2 r0 P_inv_def by (auto simp:kinetics_RSS_empty)
  next
    fix ω δt
     show ‹P_inv ω ∧ δt ∈ {0<..Δt} ⟹ ∀ ω'∈{ω'. ∀ i∈{0..N}. ω' i ∈ kinetics_RSS rss_motion i δt
ω}. P_inv ω'›
    proof(auto simp:P_inv_def)
      fix ω' i
      assume ‹∀ i∈{0..N}. ω' i ∈ kinetics_RSS rss_motion i δt ω› and ‹i ≤ N› and ‹0 < δt›
      then show ‹0 ≤ speed (ω' i)›
        apply (auto simp:k0 r0 mult.commute split:if_splits)
        by (metis as.select_convs(2) atLeastAtMost_iff less_eq_real_def not_less zero_le)
    next
      fix ω' i
      assume assm1:‹∀ i≤N. 0 ≤ speed (ω i)›
         and assm2:‹∀ i<N. d_min ω i < d_real ω i›
         and assm3:‹0 < δt›
         and assm4:‹δt ≤ Δt›
         and assm5:‹∀ i∈{0..N}. ω' i ∈ kinetics_RSS rss_motion i δt ω›
         and assm6:‹i < N›
```

```
have as1:‹0 ≤ speed (ω i)›
  using assm1[rule_format, of i] assm6 by simp
have as2:‹0 ≤ speed (ω (i+1))›
  using assm1[rule_format, of ‹i+1›] assm6 by simp
note as3 = assm2[rule_format, of i, OF assm6]
note as4 = assm3
note as5 = assm4
have as6:‹ω' i ∈ kinetics_RSS rss_motion i δt ω›
  using assm5[rule_format, of i] assm6 by simp
have as7:‹ω' (i + 1) ∈ kinetics_RSS rss_motion (i + 1) δt ω›
  using assm5[rule_format, of ‹i+1›] assm6 by simp


let ?a_r = ‹acc (ω' i)› and ?a_f = ‹acc (ω' (i+1))›
and ?v_r = ‹speed (ω i)› and ?v_f = ‹speed (ω (i+1))›
and ?v'_r = ‹speed (ω' i)› and ?v'_f = ‹speed (ω' (i+1))›
and ?p_r = ‹pos (ω i)› and ?p_f = ‹pos (ω (i+1))›
and ?p'_r = ‹pos (ω' i)› and ?p'_f = ‹pos (ω' (i+1))›
and ?b_r = ‹(speed (ω i))^2 / (2*a_minbrake)› and ?b_f = ‹(speed (ω (i+1)))^2 / (2*a_maxbrake)›

and ?b'_r = ‹(speed (ω' i))^2 / (2*a_minbrake)› and ?b'_f = ‹(speed (ω' (i+1)))^2 / (2*a_maxbrake)›


— some simplications factorized here
have simp0:‹0 ≤ ?v'_r› and simp1:‹0 ≤ ?v'_f›
  using as6 as7 by (auto simp:k0 r0 mult.commute split:if_splits)

have simp2:‹?v_r / 2 + ?a_r*δt / 2 ≥ 0› and simp3:‹?v_f / 2 + ?a_f*δt / 2 ≥ 0›
  using as1 as2 as6 as7 by (auto simp add:k0 r0 as4 mult.commute)

have simp4:‹?v_r + ?a_r*δt / 2 ≥ 0› and simp5:‹?v_f + ?a_f*δt / 2 ≥ 0›
  using as1 simp2 apply linarith
  using as2 simp3 by linarith

have simp6:‹δt*?v_r + ?a_r*δt^2 / 2 ≥ 0› and simp7:‹δt*?v_f + ?a_f*δt^2 / 2 ≥ 0›
  using as4 simp4 simp5
  by (metis (no_types, hide_lams) distrib_left less_eq_real_def mult.assoc
            mult.commute mult_nonneg_nonneg power2_eq_square times_divide_eq_right)+

have     simp4p: ‹?v_r + ?a_r*δt / 2 ≥ ?v_r / 2›
    and simp5p: ‹?v_f + ?a_f*δt / 2 ≥ ?v_f / 2›
  using as1 simp2 apply linarith
  using as2 simp3 by linarith

have     simp6p: ‹δt*?v_r + ?a_r*δt^2 / 2 ≥ δt*?v_r / 2›
    and simp7p: ‹δt*?v_f + ?a_f*δt^2 / 2 ≥ δt*?v_f / 2›
  using as4 simp4p simp5p
  by (metis distrib_left less_eq_real_def mult.left_commute mult_left_mono
          power2_eq_square times_divide_eq_right)+
```

```
have simp8:‹?v_r + a * δt ≤ 0 ⟹ − (?v_r)^2 / (a * 2) ≤ (δt*?v_r) / 2› for a
  using as1 mult_left_mono[where b=‹δt› and a=‹−?v_r / a› and c=‹?v_r›, OF _ as1]
  apply (auto simp add:mult.commute power2_eq_square split:if_splits)
  using as4 mult_pos_pos by force

have simp9:‹?v_f + a * δt ≤ 0 ⟹ − (?v_f)^2 / (a * 2) ≤ (δt*?v_f) / 2› for a
  using as2 mult_left_mono[where b=‹δt› and a=‹−?v_f / a› and c=‹?v_f›, OF _ as2]
  apply (auto simp add:mult.commute power2_eq_square split:if_splits)
  using as4 mult_pos_pos by force

— safety distance proofs
{ assume ‹?v'_f = 0›
  then have ‹max 0 ?b_f ≤ ?p'_f − ?p_f›
   using as7 as2 as4 apply (auto simp add:k0 r0 mult.commute divide_le_0_iff split:if_splits)

    using divide_left_mono[where c=‹(?v_f)^2› and a=‹(a_maxbrake * 2)›]
    by (smt (z3) zero_less_mult_iff divide_cancel_left minus_divide_right
                 mult_minus_left zero_eq_power2 zero_le_mult_iff zero_le_power)+
} note dmin_front_car_stopping=this

{  assume h0:‹?v'_f > 0›
  from as7 have h1:‹− ?a_f / a_maxbrake ≤ 1›
        by (auto simp add:k0 r0 split:if_splits) (smt (z3) a1 a2 divide_le_eq_1_neg mi-
nus_divide_right)+
  have h2:‹    (?v_f + δt * ?a_f)^2 / (2*a_maxbrake)
            =  (?v_f)^2 / (2*a_maxbrake) + (?a_f/a_maxbrake)
              * (δt * ?v_f + ?a_f * δt^2 / 2)›
    by (simp add: power2_sum distrib_left mult.commute power2_eq_square
                  add_divide_distrib mult.left_commute)
  have ‹0 ≤ ?p'_f − ?p_f›
    using as7 h0 simp7 by (auto simp add:k0 r0 mult.commute split:if_splits)
  then have ‹max 0 (?b_f − ?b'_f) ≤ ?p'_f − ?p_f›
    using as7 h0 mult_left_mono[OF h1 simp7] h2 a1 a2
    by (auto simp add:k0 r0 mult.commute split:if_splits)
} note dmin_front_car_motion=this

have front_car:‹max 0 (?b_f − ?b'_f) ≤ ?p'_f − ?p_f›
  using dmin_front_car_motion dmin_front_car_stopping simp1 by fastforce

{ assume h0:‹d_rss ω i > d_real ω i› and h1:‹?v'_r > 0›
  have h2:‹?a_r / a_minbrake ≤ −1›
    using as6 a1 a2 h0 h1 apply(auto simp add:k0 r0 split:if_splits)
    by (metis a1 add.inverse_inverse le_divide_eq_1_pos minus_divide_left neg_le_iff_le)
  have ‹?p'_r − ?p_r ≤ (?v_r)^2 / (2*a_minbrake) − (?v'_r)^2 / (2*a_minbrake)›
    using as6 a1 a2 h0 h1 mult_left_mono[OF h2 simp6]
        by (auto simp add:k0 r0 power2_sum add_divide_distrib distrib_left mult.commute
split:if_splits)
      (smt (verit) mult.assoc mult.commute power2_eq_square)
```

```
      then have ‹?b′r − ?br ≤ ?pr − ?p′r›
        by force
    } note dmin_rear_car_nonsafe_breaking=this

    { assume h0:‹drss ω i > dreal ω i› and h1:‹?v′r = 0›
      have h2:‹a ≤ −aminbrake ⟹ a / aminbrake ≤ −1› for a
        by (metis a1 divide_eq_minus_1_iff divide_le_cancel less_le not_le)
      have h4:‹?p′r − ?pr ≤ (?vr)^2 / (2∗aminbrake)›
           using as6 a1 a2 h0 h1 apply(auto simp add:k0 r0 power2_sum add_divide_distrib
split:if_splits)
        using mult_left_mono[OF h2] by (simp add: mult.commute)
      then have ‹− ?br ≤ ?pr − ?p′r›
        by argo
    } note dmin_rear_car_nonsafe_stopping=this

    have dmin_rear_car_nonsafe:‹drss ω i > dreal ω i ⟹ ?b′r − ?br ≤ ?pr − ?p′r›
     using dmin_rear_car_nonsafe_breaking dmin_rear_car_nonsafe_stopping simp0 by fastforce

    { assume h0:‹drss ω i ≤ dreal ω i› and h1:‹?v′r = 0›
      have h2: ‹(δt ∗ ?vr) / 2 ≤ ϱ∗?vr›
        using as1 as5 b less_eq_real_def by force
      have h3:‹?p′r − ?pr ≤ ϱ∗?vr›
         using as6 as1 a1 h0 h1 apply(auto simp add:d0 dmin_def k0 r0 mult.commute not_le
split:if_splits)
        using order_trans[OF simp8 h2]
        apply (smt (verit, best) mult.commute neg_divide_le_eq)
        by (metis add_cancel_right_right add_le_less_mono as4
                  mult_eq_0_iff mult_less_cancel_left_pos not_less)
      have h4: ‹?br ≤ (?vr + ϱ∗amaxaccel)^2/(2∗aminbrake)›
       by (smt (verit, ccfv_SIG) a0 a1 a3 as1 frac_le mult_pos_pos power_mono zero_le_power2)
      with h0 have h5:‹ϱ∗?vr + ?br − ?bf < ?pf − ?pr›
        by (smt (verit, best) a0 a3 d0 d1 divide_pos_pos zero_less_mult_iff zero_less_power)
      with h3 have ‹?br − ?bf < ?pf − ?p′r›
        by fastforce
    } note dmin_rear_car_safe_stopping=this

    { assume h0:‹drss ω i ≤ dreal ω i› and h1:‹?v′r > 0›
      have h2:‹δt∗?vr ≤ ϱ∗?vr›
        using as1 as4 as5 b mult_right_less_imp_less by force
      from as6 have h3:‹?ar∗δt^2/2 < amaxaccel∗ϱ^2/2›
        using a0 a1 a3 h0 apply (auto simp add:k0 r0 as4 split:if_splits)
        by (smt (verit) as4 as5 b eq_divide_imp frac_le mult.commute
                      power2_le_imp_le zero_eq_power2 zero_le_mult_iff)
      from as6 have h4: ‹?b′r ≤ (?vr + ϱ∗amaxaccel)^2/(2∗aminbrake)›
        using a1 h0 apply (auto simp add:k0 r0)
            by (smt (verit, best) a0 a1 as4 as5 b divide_le_cancel mult_mono′ power_mono
zero_le_mult_iff)
      have h5: ‹?br ≤ (?vr + ϱ∗amaxaccel)^2/(2∗aminbrake)›
       by (smt (verit, ccfv_SIG) a0 a1 a3 as1 frac_le mult_pos_pos power_mono zero_le_power2)
```

```
    from h2 h3 h4 h5 have ‹  δt∗?v_r + ?a_r∗δt²/2 + (max ?b_r ?b'_r)
                          < ϱ∗?v_r + a_maxaccel∗ϱ²/2 + (?v_r + ϱ∗a_maxaccel)²/(2∗a_minbrake)›
      by (smt (verit) mult.commute)
    with h0 have h6:‹δt∗?v_r + ?a_r∗δt²/2 + (max ?b_r ?b'_r) − ?b_f < ?p_f − ?p_r ›
      by (smt (verit, ccfv_SIG) d0 d1 mult.commute times_divide_eq_right)
    have ‹?p'_r − ?p_r = δt∗?v_r + ?a_r∗δt²/2›
   using as6 as1 a1 h1 by (auto simp add:d0 d_min_def k0 r0 mult.commute not_le split:if_splits)

    with h6 have ‹(max ?b_r ?b'_r) − ?b_f < ?p_f − ?p'_r›
      by fastforce
  } note dmin_rear_car_safe_motion=this

  have dmin_rear_car_safe:‹d_rss ω i ≤ d_real ω i ⟹ (max ?b_r ?b'_r) − ?b_f < ?p_f − ?p'_r›
    by (metis a1 divide_le_cancel less_eq_real_def max_def mult_2 power2_less_eq_zero_iff
            power_zero_numeral dmin_rear_car_safe_motion dmin_rear_car_safe_stopping
            simp0 zero_less_double_add_iff_zero_less_single_add)

  have dmin:‹?b'_r − ?b'_f < d_real ω' i›
    using as3 d1 d_min_def front_car dmin_rear_car_nonsafe dmin_rear_car_safe by fastforce

  — no collision's proofs

  { assume h0:‹?b_r ≤ ?b_f› and h1:‹?b'_r < ?b'_f›
      — ?b'0 < ?b'1, is added to profit from previous results of dmin
      and h2:‹?v'_r = 0›
    from h0 have h3:‹?v_r ≤ ?v_f›
      using mult_right_mono[OF h0, where c=‹(2 ∗ a_minbrake)›, simplified]
     by (smt (z3) a1 a2 as2 less_divide_eq mult_left_mono power2_le_imp_le zero_le_power2)

    from h2 have h6:‹?p'_r − ?p_r ≤ δt ∗ ?v_r / 2›
      using as6 as1 apply(auto simp add:k0 r0 a1)
      using simp8[simplified divide_divide_eq_left[symmetric]]
       apply (smt (verit, ccfv_threshold) mult.assoc mult.left_commute
          mult_left_mono mult_minus_left neg_le_iff_le power2_eq_square real_add_le_0_iff)
      by (metis add_cancel_left_right add_le_less_mono as4 mult_eq_0_iff
            mult_less_cancel_left_pos not_less)
    have h6:‹?p'_r − ?p_r ≤ ?p'_f − ?p_f›
      using as7 h1 h2 apply(auto simp add:k0 r0 a1)
      by (smt (verit, best) as.select_convs(3) as4 field_sum_of_halves h3 h6
                 mult.commute mult_less_cancel_left_pos simp7p)
  } note no_collision_if_slow_rear_car_stopping=this

  { assume h0:‹?b_r ≤ ?b_f› and h1:‹?b'_r < ?b'_f›
      and h2:‹?v'_r > 0›
    from h0 have h3:‹?v_r ≤ ?v_f›
      using mult_right_mono[OF h0, where c=‹(2 ∗ a_minbrake)›, simplified]
     by (smt (z3) a1 a2 as2 less_divide_eq mult_left_mono power2_le_imp_le zero_le_power2)

    have h4:‹?v'_f > 0›
```

```
    by (metis a1 div_0 divide_nonneg_pos h1 mult.commute mult_2_right
             not_less order.not_eq_order_implies_strict  simp1 zero_le_power2
             zero_less_double_add_iff_zero_less_single_add zero_power2)
  from h1 h2 h3 h4 have h5:‹?v_r + ?a_r * δt < ?v_f + ?a_f * δt›
    using as6 as7 apply(auto simp add:k0 r0)
    by (smt (verit, best) a1 a2 frac_le mult.commute power_mono zero_le_power2)
  have h6:‹δt * ?v_r + δt^2 * ?a_r < δt * ?v_f + δt^2 * ?a_f›
    by (metis (no_types, hide_lams) as4 distrib_left h5 mult.commute mult.left_commute
                             mult_less_cancel_left_pos power2_eq_square)
  with h3 have h7:‹δt * ?v_r + δt^2 * ?a_r / 2 < δt * ?v_f + δt^2 * ?a_f / 2›
    by (smt (verit, ccfv_SIG) as4 field_sum_of_halves mult_less_cancel_left_pos)
  have ‹?p'_r − ?p_r < ?p'_f − ?p_f›
    using as6 as7 h2 h4 h7
    by (auto simp add:k0 r0 a1 d_min_def d1 divide_nonneg_pos power2_sum not_le)
} note no_collision_if_slow_rear_car_nonstopping=this
```

remaining case (REM):

1. non-safe mode *dsafe* > *dreal*: in the opposite case, we profit from previous results of *dmin_rear_car_safe*

2. faster rear car : but it becomes slower after breaking

```
{ assume h0:‹?b_f < ?b_r› and h2:‹?b'_r < ?b'_f›
    and h3:‹?v'_f = 0›
  then have False
    by (metis a1 divide_eq_0_iff le_divide_eq mult_eq_0_iff not_less
             power_zero_numeral zero_le_power2 zero_less_numeral)
} note REM_front_car_not_stopping=this

{ assume h0:‹?b_f < ?b_r› and h1:‹d_rss ω i > d_real ω i› and h2:‹?v'_f > 0›
    and h3:‹?v_f − a_maxbrake *δt ≤ 0›
  have h4:‹?b_f ≤ ?p'_f − ?p_f›
    using as7 h2
 apply (auto simp add:k0 mult.commute not_le power2_sum add_divide_distrib split:if_splits)

   by (smt (z3) as.select_convs(3) divide_minus_left h3 minus_divide_right mult_minus_left
simp7p simp9)
  then have ‹0 < ?p'_f − ?p'_r›
     by (smt (verit, best) a1 as3 d1 d_min_def divide_nonneg_pos dmin_rear_car_nonsafe h1
zero_le_power2)
  } note REM_front_car_may_stop=this

{ assume h0:‹?b_f < ?b_r› and h1:‹d_rss ω i > d_real ω i› and h2:‹?v'_f > 0›
    and h3:‹?v_f − a_maxbrake *δt > 0›

 have s1:‹a_maxbrake / a_minbrake ≥ 1 ∧ sqrt a_maxbrake ≥ sqrt a_minbrake ∧ sqrt a_minbrake
> 0›
    using a1 a2 by simp
```

```
then have s2:‹a_maxbrake / a_minbrake ≥ sqrt a_maxbrake / sqrt a_minbrake›
  by (smt (verit, ccfv_threshold) divide_less_eq_1 real_div_sqrt real_sqrt_divide)
then have s3:‹a_maxbrake * sqrt a_minbrake  ≥ a_minbrake * sqrt a_maxbrake›
  by (smt (verit, ccfv_SIG) a1 a2 divide_divide_times_eq divide_le_eq_1_pos
                divide_less_eq_1 mult_nonneg_nonneg real_sqrt_gt_0_iff)

have s4:‹sqrt a_maxbrake / sqrt a_minbrake ≥ (?v_f) / (?v_r)›
  using real_sqrt_le_mono[OF order.strict_implies_order[OF h0]]
  apply(simp add:real_sqrt_divide real_sqrt_mult as1 as2 mult.commute)
   apply(subst (asm) (1 2) divide_divide_eq_left[symmetric], subst(asm) divide_le_cancel,
simp)
  by (smt (verit, ccfv_threshold) as1 divide_less_eq_1 mult.commute
                   pos_divide_le_eq s1 times_divide_eq_left)
have s5:‹?v_r / a_minbrake ≥ ?v_f / a_maxbrake›
  using order_trans[OF s4 s2] apply(subst (asm) pos_divide_le_eq)
   apply (smt (verit, ccfv_threshold) a1 a2 as1 divide_eq_0_iff divide_nonneg_pos
                     h0 zero_le_power2 zero_power2)
  apply (simp add: a1 le_divide_eq mult.commute)
  by (smt (verit) a1 a2 divide_le_cancel nonzero_mult_div_cancel_left)
then have s6:‹?v_r − a_minbrake*δt > 0›
      using h3 a1 a2 by (metis a1 diff_gt_0_iff_gt less_le_trans mult.commute
pos_less_divide_eq)

have s7:‹(?v_f) * sqrt a_minbrake ≤ (?v_r) * sqrt a_maxbrake›
  using real_sqrt_le_mono[OF order.strict_implies_order[OF h0]]
  apply(simp add:real_sqrt_divide real_sqrt_mult as1 as2 mult.commute)
   apply(subst (asm) (1 2) divide_divide_eq_left[symmetric], subst(asm) divide_le_cancel,
simp)
  apply(subst (asm) pos_divide_le_eq)
  using a1 a2 apply auto[1]
  by (simp add: a1 le_divide_eq mult.commute)
have s8:‹   (?v_f − a_maxbrake*δt) / sqrt a_maxbrake
         ≤ (?v_r − a_minbrake *δt) / sqrt a_minbrake›
  apply(subst pos_divide_le_eq) using a1 a2 apply auto
  apply(subst pos_le_divide_eq) apply (simp_all add:left_diff_distrib)
       using s7 s3 by (metis as4 diff_mono mult.commute mult.left_commute
mult_le_cancel_left_pos)
with h3 have s9:‹   (?v_f − a_maxbrake *δt)^2/(2*a_maxbrake)
                 ≤ ((?v_r − a_minbrake *δt)^2/(2*a_minbrake))›
  using abs_le_square_iff[where x=‹(?v_f − a_maxbrake*δt) / sqrt a_maxbrake›
                and y = ‹(?v_r − a_minbrake *δt) / sqrt a_minbrake›]
  using s1 apply (auto simp add:power_divide)
  by (smt (verit, best) divide_less_0_iff s1)

have s10:‹δt * ?v_r / 2 ≤ (δt * ?v_r − a_minbrake * δt^2 / 2)›
  by (smt (verit, best) as4 field_sum_of_halves le_divide_eq power2_eq_square
               real_divide_square_eq s6 zero_less_power)

have h4:‹δt*?v_f − a_maxbrake *δt^2/2 ≤ ?p'_f − ?p_f›
```

```
        using as7 h2
            apply (auto simp add:k0 r0 mult.commute not_le power2_sum add_divide_distrib
split:if_splits)
               by (metis comm_semiring_class.distrib mult_nonneg_nonneg real_0_le_add_iff
zero_le_power2)+
      have ‹δt∗?v_f − a_maxbrake ∗δt^2/2 −?b_f = −(?v_f − a_maxbrake ∗δt)^2/(2∗a_maxbrake)›
         apply(simp add:power2_diff add_divide_distrib minus_add diff_add_eq_diff_diff_swap
diff_divide_distrib)
         by (metis a1 a2 mult.assoc mult.left_commute nonzero_mult_div_cancel_left not_less
power2_eq_square)
      with h4 have h5:‹?p'_f − ?p_f − ?b_f ≥ − (?v_f − a_maxbrake ∗δt)^2/(2∗a_maxbrake)›
        by linarith

      have ‹a ≤ − a_minbrake ⟹   (δt ∗ ?v_r + a ∗ δt^2 / 2)
                                ≤ (δt ∗ ?v_r − a_minbrake ∗ δt^2 / 2)› for a
        by (smt (verit) as4 eq_divide_imp field_sum_of_halves mult_minus_left
                    pos_minus_divide_le_eq zero_less_power2)
      then have h6:‹?p'_r − ?p_r ≤  δt∗?v_r − a_minbrake ∗δt^2/2›
        using as6 h1 h2 a1 apply (auto simp add:k0 r0 d0 d1 mult.commute not_le power2_sum
add_divide_distrib min_def split:if_splits)
      using simp8 s10 by (smt (verit, ccfv_SIG) divide_le_cancel eq_divide_imp mult.commute)+

      have h7: ‹δt∗?v_r − a_minbrake ∗δt^2/2 − ?b_r = −(?v_r − a_minbrake ∗δt)^2/(2∗a_minbrake)›
        apply(simp add:power2_diff add_divide_distrib minus_add  diff_divide_distrib)
        by (metis a1 less_eq_real_def mult.assoc mult.left_commute
                  nonzero_mult_div_cancel_left not_less power2_eq_square)

      with h5 h6 s9 have ‹?p'_r − ?p_r  − ?b_r ≤ ?p'_f − ?p_f − ?b_f›
        by linarith
      then have ‹0 < ?p'_f − ?p'_r›
        using as3 d1 d_min_def by auto
    } note REM_front_car_nonstopping=this

    have nocollision:‹0 < d_real ω' i›
      apply (cases ‹?b'_r ≥ ?b'_f›)
      using dmin apply simp
      apply(cases ‹?b_r ≤ ?b_f›)
      using as3 d1 d_min_def no_collision_if_slow_rear_car_nonstopping
          no_collision_if_slow_rear_car_stopping simp0
       apply fastforce
      apply(cases ‹d_rss ω i > d_real ω i›, simp add:not_le)
                     using  REM_front_car_may_stop  REM_front_car_not_stopping  d1
REM_front_car_nonstopping simp1
       apply fastforce
      using d1 front_car dmin_rear_car_safe by force

    show ‹d_min ω' i < d_real ω' i›
      using dmin nocollision by (simp add: d_min_def)
  qed
```

```
 next
  fix ω
  have ‹P_inv ω ⟹ (i < j ∧ j ≤ N) ⟶ pos (ω i) < pos (ω j) › for i j
   unfolding d_min_def d1 P_inv_def
   apply(induct j, simp)
   by (smt (z3) Suc_leD Suc_le_lessD add.commute less_Suc_eq plus_1_eq_Suc)
  then show ‹P_inv ω ⟹ no_collision ω {0..N} 0›
   unfolding no_collision_ε_0
   by (smt (z3) atLeastAtMost_iff inj_on_def less_le not_less)
 qed
qed


end

end
```

6 An improved RSS-alike driving strategy: RSS-plus in a N-cars-Scenario

theory *RSS_plus*
imports *CSP−AutoCars*
begin

6.1 Context : modeling N cars, one lane, one direction

locale *samelane_samedirection_Ncars* =

fixes $a_{maxaccel}$::*real* **and** $a_{minbrake}$::*real* **and** $a_{maxbrake}$::*real* **and** ϱ::*real* **and** ε::*real*

assumes *a0*:⟨$0 < a_{maxaccel}$⟩
and *a1*:⟨$a_{minbrake} < 0$⟩
and *a2*:⟨$a_{maxbrake} \leq a_{minbrake}$⟩
and *a3*:⟨$\varrho > 0$⟩
and *a4*:⟨$\varepsilon > 0$⟩

begin

definition d_{real}:: ⟨*real scene* ⇒ *nat* ⇒ *real*⟩
where *d0*:⟨$d_{real}\ \omega\ i \equiv pos(\omega\ (i{+}1)) - pos(\omega\ i)$⟩

definition d_{rss}:: ⟨*real scene* ⇒ *nat* ⇒ *real*⟩
where *d1*:⟨$d_{rss}\ \omega\ i \equiv$ *let* $v_r = speed(\omega\ i)$; $v_f = speed(\omega\ (i{+}1))$ *in*
if $v_r \geq 0$
then max 0 $((\varrho{*}v_r + (\varrho^2/2){*}a_{maxaccel} + (v_r + \varrho{*}a_{maxaccel})^2/(2{*}{-}a_{minbrake}))$
$- (v_f)^2/(2{*}{-}a_{maxbrake}))$
else undefined⟩

Considering the next acceleration of the rear car where $a_r < a_{maxaccel}$

definition d_{safe}:: ⟨*real scene* ⇒ *nat* ⇒ *real* ⇒ *real*⟩
where *d2*:⟨$d_{safe}\ \omega\ i\ a_r \equiv$ *let* $v_r = speed(\omega\ i)$; $v_f = speed(\omega\ (i{+}1))$ *in*
if $v_r \geq 0$
then max 0
((*if* $v_r + \varrho{*}a_r > 0$
then $(\varrho{*}v_r + (\varrho^2/2){*}a_r + (v_r + \varrho{*}a_r)^2/(2{*}{-}a_{minbrake}))$
else $(v_r)^2/(2{*}{-}a_r))$
$- (v_f)^2/(2{*}{-}a_{maxbrake}))$
else undefined⟩

definition $drive_{rss}$::⟨*real* ⇒ *real* $motion_{acc}$⟩

where $r0$:‹$drive_{rss}\ \varepsilon'\ i\ \delta t\ \omega \equiv$ *if* $d_{rss}\ \omega\ i + \varepsilon' \leq d_{real}\ \omega\ i$
then $\{a_{maxbrake} .. a_{maxaccel}\}$
else $\{a_{maxbrake} .. a_{minbrake}\}$›

ε' is a margin we can add to drive safely

definition $drive_{safe}$::‹real ⇒ real $motion_{acc}$›
where $r1$:‹$drive_{safe}\ \varepsilon'\ i\ \delta t\ \omega \equiv \{a_{maxbrake} .. a_{maxaccel}\} \cap \{a.\ a > a_{minbrake} \longrightarrow d_{safe}\ \omega\ i\ a + \varepsilon' \leq d_{real}\ \omega\ i\}$›

Cars can not go backward! Overriding kinetics:

definition $kinetics_{fw}$:: ‹real $motion_{acc}$ ⇒ real motion›
where $k0$:‹$kinetics_{fw}\ motion_{acc} \equiv \lambda\ i\ \delta t\ \omega.$ { *if* $speed(\omega\ i) + \delta t * a > 0$
then (| $pos = pos(\omega\ i) + \delta t * speed(\omega\ i) + ((\delta t^2)/2) * a$,
$speed = speed(\omega\ i) + \delta t * a$,
$acc = a$ |)
else (| $pos = pos(\omega\ i) - (speed(\omega\ i))^2 / (2 * a)$,
$speed = 0$,
$acc = 0$ |)
$|a.\ a \in motion_{acc}\ i\ \delta t\ \omega$ }›

6.2 Preliminaries

declare Let_def[simp]

lemma $kinetics_{fw}$_empty: ‹$kinetics_{fw}\ motion_{acc}\ i\ \delta t\ \omega = \{\} \longleftrightarrow motion_{acc}\ i\ \delta t\ \omega = \{\}$›
and $kinetics_{fw}$_forward:‹$\omega'\ i \in kinetics_{fw}\ motion_{acc}\ i\ \delta t\ \omega \Longrightarrow speed(\omega'\ i) \geq 0$›
using k0 **by** auto

safety distance is monotonic w.r.t the next acceleration: when we accelerate, we increase the breaking distance

lemma d_{safe}_mono: ‹mono ($d_{safe}\ \omega\ i$)›
proof(rule monoI)
fix x::real **and** y
assume h0: ‹$x \leq y$›
have h1: ‹$0 < speed(\omega\ i) + \varrho * x \Longrightarrow 0 < speed(\omega\ i) + \varrho * y$›
by (meson a3 add_le_cancel_left h0 less_eq_real_def less_le_trans mult_left_mono)
have h2: ‹$0 < speed(\omega\ i) + \varrho * x \Longrightarrow \varrho * speed(\omega\ i) + \varrho^2 * x\ /\ 2 + (speed(\omega\ i) + \varrho * x)^2\ /\ (2 * -a_{minbrake}) \leq \varrho * speed(\omega\ i) + \varrho^2 * y/2 + (speed(\omega\ i) + \varrho * y)^2\ /\ (2 * -a_{minbrake})$›
by (smt (verit, best) a1 a3 divide_le_cancel h0 mult_eq_0_iff mult_less_cancel_left_pos power_mono zero_le_power2)
{ **assume** hh0:‹$speed(\omega\ i) \geq 0$›
and hh1:‹$0 < speed(\omega\ i) + \varrho * y$›
and hh2:‹$speed(\omega\ i) + \varrho * x \leq 0$›
have hh3:‹$-(speed(\omega\ i) + \varrho * y)^2\ /\ (2 * a_{minbrake}) \geq 0$›
using a1 divide_le_0_iff **by** fastforce
have hh4:‹$speed(\omega\ i)\ /\ 2 \leq speed(\omega\ i) + (\varrho/2) * y$›
using hh1 **by** simp
have hh5:‹$- speed(\omega\ i)/x \leq \varrho$›

```
    using hh2 hh0 by (smt (verit) a3 divide_le_0_iff neg_divide_le_eq)
   have ‹− ((speed(ω i))^2 / (2 ∗ x)) ≤ ϱ ∗ speed(ω i) + ϱ^2 ∗ y / 2 − (speed(ω i) + ϱ ∗ y)^2 / (2
∗ a_minbrake)›
    using mult_mono[OF hh5 hh4, simplified] hh3
    by (smt (verit, del_insts) a3 distrib_left divide_minus_left hh0 mult.assoc power2_eq_square
times_divide_eq_right)
  } note h3=this
  { assume hh0:‹speed(ω i) ≥ 0›
     and hh1:‹speed(ω i) + ϱ∗y ≤ 0›
     and hh2:‹speed(ω i) + ϱ∗x ≤ 0›
   have hh3:‹y ≤ 0›
    using hh0 hh1 a3
    by (metis add.commute le_add_same_cancel1 not_less order_trans zero_less_mult_iff)
   have ‹− ((speed(ω i))^2 / (2 ∗ x)) ≤ − ((speed(ω i))^2 / (2 ∗ y))›
    by (smt (verit, best) a3 divide_eq_0_iff frac_le hh0 h0 hh1 minus_divide_right zero_eq_power2
zero_le_mult_iff zero_le_power)
  } note h4=this
  show ‹d_safe ω i x ≤ d_safe ω i y›
   unfolding max_def d2 using h0 h1 h2 h3 h4 by auto
qed
```

6.3 Our motion vs RSS

```
lemma d_safe_d_rss: ‹d_rss ω i = d_safe ω i a_maxaccel›
  using d1 d2 a0 a3 by (auto simp add: le_less_trans)

lemma drive_safe_drive_rss: ‹drive_rss ε′ i δt ω ⊆ drive_safe ε′ i δt ω›
  apply(cases ‹d_rss ω i + ε′ ≤ d_real ω i›)
  using a0 a1 a2 r0 r1 apply (auto simp:d_safe_d_rss)
  by (meson add_mono_thms_linordered_semiring(3) d_safe_mono monoD order.trans)

lemma cars_safe_cars_rss:
 ‹⟦sid:kinetics_fw (drive_safe ε′)⟧ ≤ ⟦sid:kinetics_fw (drive_rss ε′)⟧›
  apply(intro allI impI μ_motion_refine)
  using a0 a1 a2 kinetics_fw_empty r0 apply auto[1]
  using drive_safe_drive_rss k0 apply auto
  using atLeastAtMost_iff by blast+

corollary ‹is_safe {0..N} (kinetics_fw (drive_safe ε′)) Δt ω_0 ε ⟹ is_safe {0..N} (kinetics_fw
(drive_rss ε′)) Δt ω_0 ε›
  by (rule is_safe_refine, rule cars_safe_cars_rss, assumption)
```

6.4 Our motion is safe

```
definition d_min:: ‹real scene ⇒ nat ⇒ real›
  where d3:‹d_min ω i ≡ let v_r = speed(ω i) ; v_f = speed(ω (i+1)) in
                      if v_r ≥ 0
                      then max 0 ((v_r)^2/(2∗−a_minbrake) − (v_f)^2/(2∗−a_maxbrake))
```

```
                    else undefined›

lemma d_min_d_safe: ‹d_min ω i = d_safe ω i a_minbrake›
  using a1 d3 d2 apply (auto simp add: power2_sum add_divide_distrib)
  by (auto simp:power2_eq_square)


declare minus_divide_le_eq[simp] not_less[simp]
theorem rss_is_safe_N_cars:
    fixes N::nat
  assumes a: ‹∀ i∈{0..N}. d_min ω_0 i + ε' ≤ d_real ω_0 i ∧ 0 ≤ speed(ω_0 i)›
      and b: ‹0 < Δt ∧ Δt ≤ ϱ›
      and c: ‹ε < ε'›
    shows ‹is_safe {0..N} (kinetics_fw (drive_safe ε')) Δt ω_0 ε›
proof -
  define P_inv where ‹P_inv ≡ λ (ω::real scene). (∀ i ≤ N. 0 ≤ speed(ω i)) ∧ (∀ i < N. d_min ω i
+ ε' ≤ d_real ω i)›

  show ?thesis
  proof(rule is_safe_rule_inv[where P_inv=‹λ ω sid. P_inv ω›])
    show ‹0 < Δt› using b by simp
  next
    show ‹P_inv ω_0›
      using a c by (auto simp add:P_inv_def d0)
  next
    fix ω δt
    have ‹∀ i∈{0..N}. {a_maxbrake..a_minbrake} ⊆ drive_safe ε' i δt ω›
      using r1 a0 a1 a2 by auto
    then show ‹P_inv ω ∧ δt ∈ {0<..Δt} ⟹ ∀ i∈{0..N}. kinetics_fw (drive_safe ε') i δt ω ≠ {}›
      using a2 kinetics_fw_empty by fastforce
  next
    fix ω δt
    show ‹P_inv ω ∧ δt ∈ {0<..Δt} ⟹ ∀ ω'∈{ω'. ∀ i∈{0..N}. ω' i ∈ kinetics_fw (drive_safe ε') i δt
ω}. P_inv ω'›
    proof(auto simp:P_inv_def)
      fix ω' i
      assume ‹∀ i∈{0..N}. ω' i ∈ kinetics_fw (drive_safe ε') i δt ω› and ‹i ≤ N› and ‹0 < δt›
      then show ‹0 ≤ speed(ω' i)›
        apply (auto simp:k0 r1 mult.commute split:if_splits)
        by (metis as.select_convs(2) atLeastAtMost_iff less_eq_real_def not_less zero_le)
    next
      fix ω' i
      assume assm1:‹∀ i≤N. 0 ≤ speed(ω i)›
         and assm2:‹∀ i<N. d_min ω i + ε' ≤ d_real ω i›
         and assm3:‹0 < δt›
         and assm4:‹δt ≤ Δt›
         and assm5:‹∀ i∈{0..N}. ω' i ∈ kinetics_fw (drive_safe ε') i δt ω›
         and assm6:‹i < N›
```

```
have as1:‹0 ≤ speed(ω i)›
  using assm1[rule_format, of i] assm6 by simp
have as2:‹0 ≤ speed(ω (i+1))›
  using assm1[rule_format, of ‹i+1›] assm6 by simp
note as3 = assm2[rule_format, of i, OF assm6]
note as4 = assm3
note as5 = assm4
have as6:‹ω' i ∈ kinetics_fw (drive_safe ε') i δt ω›
  using assm5[rule_format, of i] assm6 by simp
have as7:‹ω' (i + 1) ∈ kinetics_fw (drive_safe ε') (i + 1) δt ω›
  using assm5[rule_format, of ‹i+1›] assm6 by simp


let ?a_0 = ‹acc(ω' i)› and ?a_1 = ‹acc(ω' (i+1))›
and ?b_0 = ‹(speed(ω i))^2 / (2*−a_minbrake)› and ?b_1 = ‹(speed(ω (i+1)))^2 / (2*−a_maxbrake)›

and ?b'_0 = ‹(speed(ω' i))^2 / (2*−a_minbrake)› and ?b'_1 = ‹(speed(ω' (i+1)))^2 / (2*−a_maxbrake)›
```

some simplifying rules factorized here

```
have simp0:‹0 ≤ speed(ω' i)› and simp1:‹0 ≤ speed(ω' (i+1))›
  using as6 as7 by (auto simp:k0 r1 mult.commute split:if_splits)

have simp2:‹speed(ω i) + δt*a > 0 ⟹ speed(ω i) + δt*a/2 > speed(ω i)/2› and simp3:‹speed(ω (i+1)) + δt*a > 0 ⟹ speed(ω (i+1)) + δt*a/2 > speed(ω (i+1))/2› for a δt
  by simp+
```

— a lower bound of the traversed distance when non stopping

```
have simp4:‹δt > 0 ⟹ speed(ω i) + δt*a > 0 ⟹ δt*speed(ω i) + (δt^2/2)*a > δt*speed(ω i)/2› and simp5:‹δt > 0 ⟹ speed(ω (i+1)) + δt*a > 0 ⟹ δt*speed(ω (i+1)) + (δt^2/2)*a > δt*speed(ω (i+1))/2› for a δt
  using mult_strict_left_mono[OF simp2, of δt a δt] mult_strict_left_mono[OF simp3, of δt a δt]
  by (simp_all add: distrib_left power2_eq_square)

have simp4':‹δt > 0 ⟹ speed(ω i) + δt*a > 0 ⟹ δt*speed(ω i) + (δt^2/2)*a ≥ 0› and simp5':‹δt > 0 ⟹ speed(ω (i+1)) + δt*a > 0 ⟹ δt*speed(ω (i+1)) + (δt^2/2)*a ≥ 0› for a δt
  using as1 as2 simp4 simp5 zero_le_mult_iff by fastforce+
```

— an upper bound of the traversed distance when stopping

```
have simp6:‹δt > 0 ⟹ speed(ω i) + δt*a ≤ 0 ⟹ −(speed(ω i))^2/(2*a) ≤ δt*speed(ω i)/2› and simp7:‹δt > 0 ⟹ speed(ω (i+1)) + δt*a ≤ 0 ⟹ −(speed(ω (i+1)))^2/(2*a) ≤ δt*speed(ω (i+1))/2› for a δt
  using as1 as2 mult_left_mono[where b=‹δt› and a=‹−speed(ω i) / a› and c=‹speed(ω i)›, OF _ as1] mult_left_mono[where b=‹δt› and a=‹−speed(ω (i+1)) / a› and c=‹speed(ω (i+1))›, OF _ as2]
  apply (auto simp add:mult.commute power2_eq_square  split:if_splits)
  using real_0_le_add_iff by fastforce+
```

```
— cars are going forward
have rear_car_forward: ‹0 ≤ pos(ω' i) − pos(ω i)›
  using as6 simp4 as1 as4 a1 a2 apply (auto simp add:k0 r1 mult.commute divide_nonneg_nonpos field_sum_of_halves split:if_splits)
  apply (smt (verit, best) divide_eq_0_iff divide_nonneg_neg mult_nonneg_nonneg zero_eq_power2 zero_le_power)
  by (metis (no_types, hide_lams) mult.commute not_le simp4' times_divide_eq_right)+

have front_car_forward: ‹0 ≤ pos(ω' (i+1)) − pos(ω (i+1))›
  using as7 simp5 as2 as4 a1 apply (auto simp add:k0 r1 mult.commute divide_nonneg_nonpos field_sum_of_halves split:if_splits)
  apply (smt (verit, best) divide_eq_0_iff divide_nonneg_neg mult_nonneg_nonneg zero_eq_power2 zero_le_power)
  by (metis (no_types, hide_lams) mult.commute not_le simp5' times_divide_eq_right)+
```

6.4.1 Safety distance proofs

```
{ assume h0:‹speed(ω' (i+1)) = 0›
  then have ‹?b1 ≤ pos (ω' (i+1)) − pos (ω (i+1))›
    using a1 a2 as2 as7 as4 apply (auto simp add:k0 r1 split:if_splits)
    apply(subst times_divide_eq_right[symmetric], rule mult_left_mono[of 1 _ ‹(speed(ω (i + 1)))^2›, simplified], simp)
    apply(cases ‹speed(ω (i+1)) = 0›, simp)
    apply(subst times_divide_eq_right[symmetric], rule mult_left_mono[of 1 _ ‹(speed(ω (i + 1)))^2›, simplified])
    by (metis add_0_right add_mono_thms_linordered_field(5) divide_less_eq_1 le_divide_eq_1 mult_eq_0_iff not_le zero_less_mult_iff)
} note dmin_front_car_stopping=this

{ assume h0:‹speed(ω' (i+1)) > 0›
  from as7 have h1:‹?a1 / a_maxbrake ≤ 1›
  using a1 by (auto simp add:k0 r1) (smt (z3) a1 a2 divide_le_eq_1_neg minus_divide_right)
  have ‹(speed(ω (i+1)) + δt ∗ ?a1)^2 / (2∗a_maxbrake) = (speed(ω (i+1)))^2 / (2∗a_maxbrake) + (?a1/a_maxbrake) ∗ (δt∗speed(ω (i+1)) + (δt^2/2)∗?a1)›
    by (simp add: power2_sum distrib_left mult.commute power2_eq_square add_divide_distrib mult.left_commute)
  then have ‹?b1 − ?b'1 ≤ (pos(ω' (i+1)) − pos(ω (i+1)))›
    using h0 as7 mult_left_mono[OF h1 simp5'] as4 by (auto simp add:k0 r1 mult.commute split:if_splits)
} note dmin_front_car_motion=this

have front_car:‹?b1 − ?b'1 ≤ pos (ω' (i+1)) − pos (ω (i+1))›
  using dmin_front_car_motion dmin_front_car_stopping simp1 by fastforce

{ fix a
  assume h0:‹speed(ω' i) = 0›
  assume h1:‹a ≤ a_minbrake› and h2:‹pos (ω' i) − pos (ω i) = (speed(ω i))^2/(2∗−a)›
  have h3:‹(speed(ω i))^2/(2∗−a) ≤ (speed(ω i))^2/(2∗−a_minbrake)›
```

```
      using a1 a2 h1 by (smt (z3) frac_le zero_le_power2)
    with h2 have ‹pos (ω' i) − pos (ω i) ≤ ?b₀›
      by linarith
  } note dmin_rear_car_stopping_after_breaking=this

  { fix a
    assume h0:‹speed(ω' i) = 0›
    assume h1:‹a > a_minbrake› and h2:‹d_safe ω i a + ε' ≤ d_real ω i› and h3:‹pos (ω' i) − pos
(ω i) = (speed(ω i))²/(2∗−a)› and h4:‹speed(ω i) + δt∗a ≤ 0›
    have h5:‹speed(ω i) + ϱ∗a ≤ 0›
      by (smt (verit, best) as1 assm3 assm4 b h4 mult_le_cancel_right mult_pos_pos)
    have ‹pos(ω' i) − pos(ω i) − ?b₁ + ε' ≤ pos(ω (i+1)) − pos(ω i)›
      using h2 h5 apply (auto simp add:k0 r1 d0 d2 as1 max_def split:if_splits)
      using c h0 h3 by force+
  } note dmin_rear_car_stopping_after_slowing=this

  have dmin_rear_car_stopping:‹speed(ω' i) = 0 ⟹ − ?b₁ + ε' ≤ pos(ω (i+1)) − pos(ω' i)›
  proof −
    assume h0:‹speed(ω' i) = 0›
    then obtain a where h1:‹a ≤ a_minbrake ∨ d_safe ω i a + ε' ≤ d_real ω i› and h3:‹pos (ω'
i) − pos (ω i) = (speed(ω i))²/(2∗−a)› and h4:‹speed(ω i) + δt∗a ≤ 0›
      using as6 apply(auto simp add:k0 r1 a1 split:if_splits)
      apply blast apply blast
      using a1 apply linarith+
      by fastforce
    show ?thesis
      apply(cases ‹a ≤ a_minbrake›)
      using as1 as3 dmin_rear_car_stopping_after_breaking h0 h3 d0 d3 apply fastforce
      using dmin_rear_car_stopping_after_slowing h0 h1 h3 h4 by force
  qed

  { assume h0:‹speed(ω' i) > 0› and h1:‹?a₀ ≤ a_minbrake› — h0 is not necessary, it is induced
by h1
    have h2:‹?a₀/a_minbrake ≥ 1›
      using a1 a2 h1 by auto
    with h1 have ‹pos (ω' i) − pos (ω i) ≤ ?b₀ − ?b'₀›
     using as6 a1 a2 h1 apply (auto simp add:k0 r1 power2_sum add_divide_distrib distrib_left

                              not_le split:if_splits)
      by (drule mult_left_mono[OF h2 simp4'[OF as4], simplified], simp add:distrib_right
              distrib_left power2_eq_square mult.left_commute add_divide_distrib)+
  } note dmin_rear_car_breaking_without_stopping=this

  { assume h0:‹speed(ω' i) > 0› and h1:‹?a₀ > a_minbrake› and h2:‹?a₀ ≥ 0›
    have h3:‹speed(ω' i) = speed(ω i) + ?a₀∗δt› and h4:‹speed(ω i) + ?a₀∗δt > 0›
      using h0 as6 by (auto simp:k0 r1 mult.commute)
    then have h5:‹speed(ω i) + ?a₀∗ϱ > 0›
      by (smt (verit) a3 as1 h2 mult_nonneg_nonneg zero_less_mult_iff)
    from h0 h1 have h6:‹d_safe ω i ?a₀ + ε' ≤ d_real ω i›
```

```
      using as6 k0 r1 apply(simp split:if_splits) by auto
    have h7:‹ δt*speed(ω i) + (δt²/2)*?a₀ + (speed(ω i) + δt*?a₀)² / (2*−a_minbrake)
            ≤ ϱ*speed(ω i) + (ϱ²/2)*?a₀ + (speed(ω i) + ϱ*?a₀)² / (2*−a_minbrake)› (is ‹?A›)
    proof−
      have h8:‹0 < 1 + ?a₀ / −a_minbrake›
        using a1 a2 h1 by auto
        then have h9:‹0 ≤ speed(ω i) + ?a₀*(ϱ+δt)/2 + (?a₀/−a_minbrake)*(speed(ω i) +
?a₀*(ϱ+δt)/2)›
          using mult_pos_pos[OF h8, of ‹speed(ω i) + ?a₀*(ϱ+δt)/2›, simplified distrib_right]
        by (smt (z3) distrib_left field_sum_of_halves h4 h5)
      have h10:‹ (ϱ − δt)*(speed(ω i) + (ϱ+δt)*?a₀/2)
              = (ϱ*speed(ω i) − δt*speed(ω i)) + (ϱ²/2*?a₀ − δt²/2*?a₀)›
        apply(subst distrib_left, subst left_diff_distrib, simp)
        by (metis (no_types, lifting) mult.assoc mult.commute power2_eq_square
                                 right_diff_distrib' square_diff_square_factored)
      have h11:‹ (ϱ − δt)*(?a₀/−a_minbrake)*(speed(ω i) + (ϱ+δt)*?a₀/2)
                 = (speed(ω i) + ϱ*?a₀)² / (2*−a_minbrake) − (speed(ω i) + δt*?a₀)² /
(2*−a_minbrake)›
        apply(subst diff_divide_distrib[symmetric])
        apply(simp add:power2_eq_square square_diff_square_factored )
        apply(subst divide_divide_eq_left[symmetric],
             subst (7) mult.commute, subst left_diff_distrib, subst divide_cancel_right)
        using a1 by (simp add: distrib_right)
      have ‹0 ≤ (ϱ*speed(ω i) − δt*speed(ω i)) +
              (ϱ²/2*?a₀ − δt²/2*?a₀) +
           ((speed(ω i) + ϱ*?a₀)²/(2*−a_minbrake) − (speed(ω i) + δt*?a₀)²/(2*−a_minbrake))›
        apply(subst h10[symmetric], subst h11[symmetric])
        using mult_nonneg_nonneg[OF _ h9, of ‹ϱ − δt›]
        by (metis (no_types, hide_lams) assm4 b diff_ge_0_iff_ge distrib_left
                                  le_less_trans mult.assoc mult.commute not_less)
      then show ?A
        by simp
    qed
    with h3 h4 h5 have ‹pos (ω' i) − pos (ω i) + ?b'₀ − ?b₁ + ε' ≤ pos (ω (i+1)) − pos (ω i)›
      using as6 as1 a1 h1 by (auto simp add:d0 d2 k0 r1 mult.commute not_le split:if_splits)
  } note dmin_rear_car_safe_accelerating=this

  { assume   h0:‹speed(ω' i) > 0› and h1:‹?a₀ > a_minbrake›
        and h2:‹?a₀ < 0› and h3:‹speed(ω i) + ?a₀*ϱ > 0›
    have h4:‹speed(ω' i) = speed(ω i) + ?a₀*δt› and h5:‹speed(ω i) + ?a₀*δt > 0›
      using h0 as6 by (auto simp:k0 r1 mult.commute)
    from h0 h1 have h6:‹d_safe ω i ?a₀ + ε' ≤ d_real ω i›
      using as6 k0 r1 apply(simp split:if_splits) by auto
    have h7:‹ δt*speed(ω i) + (δt²/2)*?a₀ + (speed(ω i) + δt*?a₀)² / (2*−a_minbrake)
            ≤ ϱ*speed(ω i) + (ϱ²/2)*?a₀ + (speed(ω i) + ϱ*?a₀)² / (2*−a_minbrake)› (is ‹?A›)
    proof−
      have h8:‹0 < 1 + ?a₀ / −a_minbrake›
        using a1 a2 h1 by auto
        then have h9:‹0 ≤ speed(ω i) + ?a₀*(ϱ+δt)/2 + (?a₀/−a_minbrake)*(speed(ω i) +
```

```
?a₀*(ϱ+δt)/2)›
          using mult_pos_pos[OF h8, of ‹speed(ω i) + ?a₀*(ϱ+δt)/2›, simplified distrib_right]
          by (smt (z3) distrib_left field_sum_of_halves h3 h5)
        have h10:‹  (ϱ − δt)*(speed(ω i) + (ϱ+δt)*?a₀/2)
                  = (ϱ*speed(ω i) − δt*speed(ω i)) + (ϱ²/2*?a₀ − δt²/2*?a₀)›
          apply(subst distrib_left, subst left_diff_distrib, simp)
          by (metis (no_types, lifting) mult.assoc mult.commute power2_eq_square
                            right_diff_distrib' square_diff_square_factored)
        have h11:‹ (ϱ − δt)*(?a₀/−a_minbrake)*(speed(ω i) + (ϱ+δt)*?a₀/2)
                  = (speed(ω i) + ϱ*?a₀)² / (2*−a_minbrake) − (speed(ω i) + δt*?a₀)² /
(2*−a_minbrake)›
          apply(subst diff_divide_distrib[symmetric])
          apply(simp add:power2_eq_square square_diff_square_factored )
          apply(subst divide_divide_eq_left[symmetric], subst (7) mult.commute,
               subst left_diff_distrib, subst divide_cancel_right)
          using a1 by (simp add: distrib_right)
        have ‹0 ≤ (ϱ*speed(ω i) − δt*speed(ω i)) +
                 (ϱ²/2*?a₀ − δt²/2*?a₀) +
             ((speed(ω i) + ϱ*?a₀)²/(2*−a_minbrake) − (speed(ω i) + δt*?a₀)²/(2*−a_minbrake))›
          apply(subst h10[symmetric], subst h11[symmetric])
          using mult_nonneg_nonneg[OF _ h9, of ‹ϱ − δt›]
          by (metis (no_types, hide_lams) assm4 b diff_ge_0_iff_ge distrib_left
                            le_less_trans mult.assoc mult.commute not_less)
        then show ?A
          by simp
      qed
      with h3 h4 h5 have ‹pos (ω' i) − pos (ω i) + ?b'₀ − ?b₁ + ε' ≤ pos (ω (i+1)) − pos (ω i)›
        using as6 as1 a1 h1 by (auto simp add:d0 d2 k0 r1 mult.commute not_le split:if_splits)
    } note dmin_rear_car_safe_slowing_no_possibly_stopping=this
```

— *dmin_rear_car_safe_accelerating* and *dmin_rear_car_safe_slowing_no_possibly_stopping* can be merged

```
    { assume    h0:‹speed(ω' i) > 0› and h1:‹?a₀ > a_minbrake›
            and h2:‹?a₀ < 0› and h3:‹speed(ω i) + ?a₀*ϱ ≤ 0›
      from h0 h1 have h4:‹d_safe ω i ?a₀ + ε' ≤ d_real ω i›
        using as6 k0 r1 apply(simp split:if_splits) by auto
      have h5:‹speed(ω i) + ?a₀*δt > 0› and h6:‹speed(ω' i) = speed(ω i) + ?a₀*δt›
        using h0 as6 as1 as4 by (auto simp:k0 r1 mult.commute)
      have ‹   δt*speed(ω i) + (δt²/2)*?a₀ + (speed(ω i) + δt*?a₀)² / (2*−a_minbrake)
            ≤ (speed(ω i))² / (2*−?a₀)› (is ‹?A›)
      proof −
        from h2 have h7:‹  δt*speed(ω i) + (δt²/2)*?a₀ + (speed(ω i) + δt*?a₀)² / (2*−?a₀)
                  = (speed(ω i))² / (2*−?a₀)›
          apply (simp add:power2_sum add_divide_distrib)
          by (smt (z3) eq_divide_imp mult.assoc mult.commute power2_eq_square)
        show ?A
          using h1 h2 h7 h5 h6
          by (smt (z3) frac_le zero_le_power2)
```

```
    qed
    with h2 h3 have ‹pos (ω' i) − pos (ω i) + ?b'₀ − ?b₁ + ε' ≤ pos (ω (i+1)) − pos (ω i) ›
      using as6 as1 a1 h1 by (auto simp add:d0 d2 k0 r1 mult.commute not_le split:if_splits)
  } note dmin_rear_car_safe_slowing_possibly_stopping=this

  have dmin_rear_car_safe_motion:‹speed(ω' i) > 0 ⟹ ?a₀ > a_minbrake ⟹
                              ?b'₀ − ?b₁ + ε' ≤pos (ω (i+1)) − pos (ω' i)›
   using dmin_rear_car_safe_accelerating dmin_rear_car_safe_slowing_no_possibly_stopping

        dmin_rear_car_safe_slowing_possibly_stopping by force

  have dmin:‹?b'₀ − ?b'₁ + ε' ≤ d_real ω' i›
    apply(cases ‹speed(ω' i) = 0›)
    using d0 dmin_rear_car_stopping front_car apply simp
    apply(cases ‹?a₀ ≤ a_minbrake›)
    using as1 as3 d0 d3 dmin_rear_car_breaking_without_stopping
        dmin_rear_car_safe_motion front_car simp0 by auto
```

6.4.2 "No collision" proofs when $?b'_0 < ?b'_1$

```
  { assume h0:‹?b₀ ≤ ?b₁› and h1:‹?b'₀ < ?b'₁› and h2:‹speed(ω' i) = 0›
    from h0 have h3:‹speed(ω i) ≤ speed(ω (i+1))›
      using mult_right_mono[OF h0, where c=‹(2∗−a_minbrake)›, simplified]
     by (smt (z3) a1 a2 as2 less_divide_eq mult_left_mono power2_le_imp_le zero_le_power2)
    from h2 have h4:‹pos(ω' i) − pos(ω i) ≤ δt∗speed(ω i) / 2›
      using as6 as1 simp6[OF as4] by(auto simp add:k0 r1)
    have h5:‹speed(ω' (i+1)) > 0›
      using h1 h2 not_le simp1 by fastforce
    then have h6:‹δt∗speed(ω (i+1)) / 2 < pos (ω' (i+1)) − pos (ω (i+1))›
      using as7 h1 h2 simp5[OF as4] by (auto simp add:k0 r1 a1)
    have ‹pos (ω' i) − pos (ω i) ≤ pos (ω' (i+1)) − pos (ω (i+1))›
     using as4 h3 h4 h6 by (smt (verit, ccfv_SIG) divide_le_cancel mult_less_cancel_left_pos)
  } note no_collision_if_slow_rear_car_stopping=this

  { assume h0:‹?b₀ ≤ ?b₁› and h1:‹?b'₀ < ?b'₁› and h2:‹speed(ω' i) > 0›
    from h0 have h3:‹speed(ω i) ≤ speed(ω (i+1))›
      using mult_right_mono[OF h0, where c=‹(2 ∗−a_minbrake)›, simplified]
     by (smt (z3) a1 a2 as2 less_divide_eq mult_left_mono power2_le_imp_le zero_le_power2)

    have h4:‹speed(ω' (i+1)) > 0›
      by (smt (z3) a1 a2 frac_le h1 simp1 zero_le_power2 zero_power2)
    have ‹(speed(ω' i))²/(2∗−a_maxbrake) < ?b'₁›
      by (smt (verit, ccfv_SIG) a1 a2 frac_le h1 zero_le_power2)
    with h1 h2 h4 have h5:‹speed(ω i) + ?a₀∗δt < speed(ω (i+1)) + ?a₁∗δt›
      using as6 as7 apply(auto simp:k0 r1)
      by (smt (verit, ccfv_SIG) a1 a2 divide_le_cancel mult.commute
                        power2_le_imp_le power_mono zero_less_numeral)+
    have h6:‹δt∗speed(ω i) + δt²∗?a₀ < δt∗speed(ω (i+1)) + δt²∗?a₁›
      by (metis (no_types, hide_lams) as4 distrib_left h5 mult.commute mult.left_commute
```

```
                          mult_less_cancel_left_pos power2_eq_square)
    with h3 have h7:‹δt∗speed(ω i) + δt²∗?a₀/2 < δt∗speed(ω (i+1)) + δt²∗?a₁/2›
      by (smt (verit, ccfv_SIG) as4 field_sum_of_halves mult_less_cancel_left_pos)
    have ‹pos (ω' i) − pos (ω i) < pos (ω' (i+1)) − pos (ω (i+1))›
      using as6 as7 h2 h4 h7
      by (auto simp add:k0 r1 a1 d3 d1 divide_nonneg_pos power2_sum not_le)
  } note no_collision_if_slow_rear_car_nonstopping=this

  { assume h1:‹?b'₀ < ?b'₁› and h2:‹speed(ω (i+1)) + a_maxbrake ∗δt ≤ 0›
        — h1 only avoids the case when the front car stops
    have h3:‹speed(ω' (i+1)) > 0›
      by (smt (verit) a1 a2 as1 divide_eq_0_iff frac_le h1 simp1
                zero_le_power2 zero_power2 divide_nonneg_pos)+
    have h4:‹?b₁ ≤ pos (ω' (i+1)) − pos (ω (i+1))›
      using as7 h2 h3 a1 a2
        apply (auto simp add:k0 r1 mult.commute not_le power2_sum add_divide_distrib
split:if_splits)
      using simp5[OF as4] simp7[OF as4]
      by (smt (verit, del_insts) divide_minus_left minus_divide_le_eq
                     mult.commute mult_less_0_iff times_divide_eq_right)+
    then have ‹ε' ≤ pos (ω' (i+1)) − pos (ω' i)›
      using a1 simp0 as1 as3[simplified d0 d3] dmin_rear_car_breaking_without_stopping
         dmin_rear_car_safe_motion dmin_rear_car_stopping
      by (smt (z3) divide_le_0_iff zero_less_power)
  } note no_collision_if_front_car_may_stop_but_dont=this

  { assume h0:‹?b₁ < ?b₀› and h1:‹?a₀ ≤ a_minbrake ∨ speed(ω' i) = 0› and h2:‹speed(ω (i+1))
+ a_maxbrake ∗δt > 0›

    have h3:‹speed(ω i) > 0›
      by (smt (verit, ccfv_threshold) a1 a2 as1 divide_eq_0_iff
                          divide_nonneg_pos h0 zero_le_power2 zero_power2)

      have s1:‹−a_maxbrake/−a_minbrake ≥ 1 ∧ sqrt(−a_maxbrake) ≥ sqrt(−a_minbrake) ∧
sqrt(−a_minbrake) > 0›
      using a1 a2 by simp
    then have s2:‹a_maxbrake/a_minbrake ≥ sqrt(−a_maxbrake) / sqrt(−a_minbrake)›
      by (smt (z3) divide_le_eq_1_pos minus_divide_divide real_div_sqrt real_sqrt_divide)
    then have s3:‹a_maxbrake∗sqrt(−a_minbrake) ≤ a_minbrake∗sqrt(−a_maxbrake)›
          by (smt (verit, best) a1 le_divide_eq mult.commute mult_minus_right s1
times_divide_eq_right)

    have s4:‹sqrt(−a_maxbrake) / sqrt(−a_minbrake) ≥ (speed(ω (i+1)))/(speed(ω i))›
      using real_sqrt_le_mono[OF order.strict_implies_order[OF h0],
                     simplified real_sqrt_divide real_sqrt_mult
                      mult.commute divide_divide_eq_left[symmetric] divide_le_cancel]
      apply(simp add: as1 as2) apply(rule mult_imp_div_pos_le)
      by (simp add: h3)
        (smt (verit) as1 divide_le_eq le_divide_eq mult.commute s1 times_divide_eq_right)
```

```
    have s5:‹speed(ω i)/−a_minbrake ≥ speed(ω (i+1))/−a_maxbrake›
      using order_trans[OF s4 s2] apply(subst (asm) pos_divide_le_eq)
       apply (smt (verit, ccfv_threshold) a1 a2 as1 divide_eq_0_iff
                                    divide_nonneg_pos h0 zero_le_power2 zero_power2)
      apply (simp add: a1 a2 le_divide_eq mult.commute)
      by (smt (verit, ccfv_SIG) a1 a2 mult.commute)
    have ‹speed(ω (i+1))/−a_maxbrake > δt›
      apply(rule mult_imp_less_div_pos)
      using a1 a2 apply fastforce
      by (metis diff_gt_0_iff_gt diff_minus_eq_add h2 mult.commute mult_minus_right)
    with s5 have s6:‹speed(ω i) + a_minbrake*δt > 0›
      using h2 a1 a2
      by (smt (z3) le_divide_eq minus_divide_right mult.commute mult_minus_right)

    have s7:‹(speed(ω (i+1)))*sqrt (−a_minbrake) ≤ (speed(ω i))*sqrt (−a_maxbrake)›
      using s4 by (metis divide_le_eq h3 le_divide_eq mult.commute s1 times_divide_eq_left)
         have s8:‹(speed(ω (i+1)) + a_maxbrake*δt)/sqrt(−a_maxbrake) ≤ (speed(ω i) +
a_minbrake*δt)/sqrt(−a_minbrake)›
      apply(subst pos_divide_le_eq) using a1 a2 apply auto
      apply(subst pos_le_divide_eq) apply (simp_all add:distrib_right)
      using s7 s3
      by (smt (verit, del_insts) add_mono_thms_linordered_semiring(1)
                             assm3 mult.commute mult.left_commute mult_le_cancel_left_pos)
    with h3 have s9:‹  (speed(ω (i+1)) + a_maxbrake*δt)^2/(2*−a_maxbrake)
                  ≤ ((speed(ω i) + a_minbrake *δt)^2/(2*−a_minbrake))›
     using abs_le_square_iff[where x=‹(speed(ω (i+1)) + a_maxbrake*δt)/sqrt(−a_maxbrake)›

                             and y = ‹(speed(ω i) + a_minbrake*δt)/sqrt(−a_minbrake)›]
      using s1 apply (auto simp add:power_divide)
      using h2 s6 by linarith

    have h4:‹δt*speed(ω (i+1)) + a_maxbrake*δt^2/2 ≤ pos(ω' (i+1)) −pos (ω (i+1))›
      using as7 h2 assm3
          apply (auto simp add:k0 r1 mult.commute not_le power2_sum add_divide_distrib
split:if_splits)
     using a1 by (smt (z3) assm3 mult.commute mult_less_cancel_left_pos zero_less_mult_iff)+
        have ‹δt*speed(ω (i+1)) + a_maxbrake *δt^2/2 −?b_1 = −(speed(ω (i+1)) + a_maxbrake
*δt)^2/(2*−a_maxbrake)›
      using s1 apply(auto simp add:power2_diff add_divide_distrib minus_add
                               diff_add_eq_diff_diff_swap diff_divide_distrib power2_sum)
      by (simp add: power2_eq_square)
    with h4 have h5:‹  pos(ω' (i+1)) − pos(ω (i+1)) − ?b_1
                  ≥ −(speed(ω (i+1)) + a_maxbrake*δt)^2/(2*−a_maxbrake)›
      by linarith

   have h6':‹a ≤ a_minbrake ⟹ (δt*speed(ω i) + a*δt^2 / 2) ≤ δt*speed(ω i) + a_minbrake*δt^2
/ 2› for a
               by (smt (verit) as4 h1 eq_divide_imp field_sum_of_halves mult_minus_left
pos_minus_divide_le_eq zero_less_power2)
```

```
    have ‹speed (ω i) + a*δt ≤ 0 ⟹ (speed(ω i))^2/(2*−a) ≤ δt*speed(ω i)/2› for a
      using simp6[OF as4] by (smt (z3) divide_minus_left minus_divide_right mult.commute)
      then have h6'':‹speed (ω i) + a*δt ≤ 0 ⟹ (speed(ω i))^2/(2*−a) ≤ δt*speed(ω i) +
a_minbrake*δt^2 / 2› for a
      using simp4[OF as4, of a_minbrake] s6
      by (smt (verit) mult.commute times_divide_eq_right)
    from h6' h6'' have h6:‹pos(ω' i) − pos(ω i) ≤  δt*speed(ω i) + a_minbrake*δt^2/2›
      using as6 a1 h1 h3 apply (auto simp add:k0 r1 d0 mult.commute split:if_splits)
      by (smt (z3) mult.commute)+

        have h7: ‹δt*speed(ω i) + a_minbrake*δt^2/2 − ?b_0 = −(speed(ω i) + a_minbrake
*δt)^2/(2*−a_minbrake)›
      using s1 apply(auto simp add:power2_diff add_divide_distrib minus_add
                            diff_add_eq_diff_diff_swap diff_divide_distrib power2_sum)
      by (simp add: power2_eq_square)

    with h5 h6 s9 have ‹pos(ω' i) − pos(ω i)  − ?b_0 ≤ pos(ω' (i+1)) − pos(ω (i+1)) − ?b_1›

      by linarith
    then have ‹ε' ≤ pos (ω' (i+1)) − pos (ω' i)›
      using as1 as3 d0 d3 by auto
  } note no_collision_if_fast_rear_car_breaking_or_stopping_and_front_car_may_not_stop=this
— 2 cases

   { assume h0:‹?b_1 < ?b_0› and h1:‹?a_0 > a_minbrake› and h2:‹speed(ω (i+1)) + a_maxbrake *δt
> 0› and h3:‹speed(ω' i) > 0›

    have h3':‹speed(ω i) > 0›
          by (smt (verit, ccfv_threshold) a1 a2 as1 divide_eq_0_iff divide_nonneg_pos h0
zero_le_power2 zero_power2)

       have s1:‹−a_maxbrake/−a_minbrake ≥ 1 ∧ sqrt(−a_maxbrake) ≥ sqrt(−a_minbrake) ∧
sqrt(−a_minbrake) > 0›
      using a1 a2 by simp
    then have s2:‹a_maxbrake/a_minbrake ≥ sqrt(−a_maxbrake) / sqrt(−a_minbrake)›
      by (smt (z3) divide_le_eq_1_pos minus_divide_divide real_div_sqrt real_sqrt_divide)
    then have s3:‹a_maxbrake*sqrt(−a_minbrake) ≤ a_minbrake*sqrt(−a_maxbrake)›
            by (smt (verit, best) a1 le_divide_eq mult.commute mult_minus_right s1
times_divide_eq_right)

    have s4:‹sqrt(−a_maxbrake) / sqrt(−a_minbrake) ≥ (speed(ω (i+1)))/(speed(ω i))›
     using real_sqrt_le_mono[OF order.strict_implies_order[OF h0], simplified real_sqrt_divide
real_sqrt_mult mult.commute divide_divide_eq_left[symmetric] divide_le_cancel]
      apply(simp add: as1 as2) apply(rule mult_imp_div_pos_le)
      by (simp add: h3')
        (smt (verit) as1 divide_le_eq le_divide_eq mult.commute s1 times_divide_eq_right)
    have s5:‹speed(ω i)/−a_minbrake ≥ speed(ω (i+1))/−a_maxbrake›
      using order_trans[OF s4 s2] apply(subst (asm) pos_divide_le_eq)
        apply (smt (verit, ccfv_threshold) a1 a2 as1 divide_eq_0_iff divide_nonneg_pos h0
```

```
zero_le_power2 zero_power2)
      apply (simp add: a1 a2 le_divide_eq mult.commute)
      by (smt (verit, ccfv_SIG) a1 a2 mult.commute)
    have ‹speed(ω (i+1))/−a_maxbrake > δt›
      apply(rule mult_imp_less_div_pos)
      using a1 a2 apply fastforce
      by (metis diff_gt_0_iff_gt diff_minus_eq_add h2 mult.commute mult_minus_right)
    with s5 have s6:‹speed(ω i) + a_minbrake*δt > 0›
      using h2 a1 a2
      by (smt (z3) le_divide_eq minus_divide_right mult.commute mult_minus_right)
    have s6':‹speed (ω i) + δt*?a_0 > speed(ω i) + a_minbrake*δt›
      by (simp add: assm3 h1)

    have s7:‹(speed(ω (i+1)))*sqrt (−a_minbrake) ≤ (speed(ω i))*sqrt (−a_maxbrake)›
     using s4 by (metis divide_le_eq h3' le_divide_eq mult.commute s1 times_divide_eq_left)
        have s8:‹(speed(ω (i+1)) + a_maxbrake*δt)/sqrt(−a_maxbrake) ≤ (speed(ω i) + a_minbrake*δt)/sqrt(−a_minbrake)›
      apply(subst pos_divide_le_eq) using a1 a2 apply auto
      apply(subst pos_le_divide_eq) apply (simp_all add:distrib_right)
      using s7 s3
       by (smt (verit, del_insts) add_mono_thms_linordered_semiring(1) assm3 mult.commute mult.left_commute mult_le_cancel_left_pos)
     then have s9:‹(speed(ω (i+1)) + a_maxbrake*δt)^2/(2*−a_maxbrake) ≤ ((speed(ω i) + a_minbrake *δt)^2/(2*−a_minbrake))›
     using abs_le_square_iff[where x=‹(speed(ω (i+1)) + a_maxbrake*δt)/sqrt(−a_maxbrake)› and y = ‹(speed(ω i) + a_minbrake*δt)/sqrt(−a_minbrake)›]
      using s1 apply (auto simp add:power_divide)
      using h2 s6 by linarith

    have h4:‹δt*speed(ω (i+1)) + a_maxbrake*δt^2/2 ≤ pos(ω' (i+1)) −pos (ω (i+1))›
        using as7 h2 assm3 apply (auto simp add:k0 r1 mult.commute not_le power2_sum add_divide_distrib split:if_splits)
   using a1 by (smt (z3) assm3 mult.commute mult_less_cancel_left_pos zero_less_mult_iff)+
     have ‹δt*speed(ω (i+1)) + a_maxbrake *δt^2/2 −?b_1 = −(speed(ω (i+1)) + a_maxbrake *δt)^2/(2*−a_maxbrake)›
             using s1 apply(auto simp add:power2_diff add_divide_distrib minus_add diff_add_eq_diff_diff_swap diff_divide_distrib power2_sum)
      by (simp add: power2_eq_square)
         with h4 have h5:‹pos(ω' (i+1)) − pos(ω (i+1)) − ?b_1 ≥ −(speed(ω (i+1)) + a_maxbrake*δt)^2/(2*−a_maxbrake)›
      by linarith

    have h7: ‹− ?b'_0 ≤ −(speed(ω i) + a_minbrake *δt)^2/(2*−a_minbrake)›
      using as6 a1 h1 h2 h3 apply (auto simp add:k0 r1 split:if_splits)
      apply(rule divide_right_mono_neg)
      using s6 s6' apply force
      by linarith

    with h5 s9 have h8:‹− ?b'_0 ≤ pos(ω' (i+1)) − pos(ω (i+1)) − ?b_1›
```

```
        by linarith

      have h9:‹pos (ω' i) − pos (ω i) + ?b'₀ − ?b₁ + ε' ≤ pos (ω (i+1)) − pos (ω i)›
        using dmin_rear_car_safe_motion dmin_rear_car_stopping h1 simp0 by force

      with h8 have ‹ε' ≤ pos (ω' (i+1)) − pos (ω' i)›
        by linarith
  } note no_collision_if_fast_rear_car_not_breaking_not_stopping_and_front_car_may_not_stop=this

    have nocollision:‹ε' ≤ d_real ω' i›
      apply (cases ‹?b'₀ ≥ ?b'₁›)
      using dmin apply simp
      apply(cases ‹?b₀ ≤ ?b₁›)
                using   as3   as1   d0   d3   simp0   no_collision_if_slow_rear_car_nonstopping
no_collision_if_slow_rear_car_stopping apply force
      apply(cases ‹?a₀ > a_minbrake›)
    using d0 no_collision_if_fast_rear_car_breaking_or_stopping_and_front_car_may_not_stop

          no_collision_if_fast_rear_car_not_breaking_not_stopping_and_front_car_may_not_stop

              no_collision_if_front_car_may_stop_but_dont simp0 apply force
    using d0 no_collision_if_fast_rear_car_breaking_or_stopping_and_front_car_may_not_stop

              no_collision_if_front_car_may_stop_but_dont by force

    show ‹d_min ω' i + ε' ≤ d_real ω' i›
      using dmin nocollision by (simp add:simp0 d3)
  qed
 next
  fix ω
  have ‹P_inv ω ⟹ (i < j ∧ j ≤ N) ⟶ pos (ω i) + ε' ≤ pos (ω j) › for i j
    unfolding d3 d0 P_inv_def
    apply (induct j, simp)
    by (smt (z3) Suc_leD Suc_le_lessD a4 add.commute c less_Suc_eq plus_1_eq_Suc)
    with a4 c show ‹P_inv ω ⟹ no_collision ω {0..N} ε›
    unfolding no_collision_def apply auto
    by (smt (verit, ccfv_threshold) le_neq_implies_less not_le)
 qed
qed

end

end
```

7 Conclusion

7.1 Summary

7.1.1 Outline of the Theory Development

This deliverable consists of an integrated document used for the generation of this pdf-document. Validation and generation require an Isabelle2021 system https://isabelle.in.tum.de/ (see also: https://en.wikipedia.org/wiki/Isabelle_(proof_assistant) for an introduction). The Isabelle system is open-source. The theory graph of this integrated document is shown in Figure 7.1. The shown theories belong to the following ccategories:

1. Isabelle System (open source): *Pure*, `Tools`, `HOL`, `HOL-Library`, `HOL-Analysis`, *HOLCF*, `AFP.Regular-Sets`, `AFP.Functional-Automata` , `HOL-Eisbach`
2. Background (Paris-Saclay): *HOL−CSP.CSP*, *CSP_RefTK.CSP_ext*, *CSP_RefTK.CSP_ext*, *Isabelle_DOF.technical_report*,
3. Foreground (Sub-Contract SystemX-SVR): `Frontmatter`, `Introduction`, *CSP_AV.CSP−AutoCars*, *CSP_AV.MOSAR−Encodings*, *CSP_AV.RSS_2cars*, *CSP_AV.RSS_Ncars*, *CSP_AV.RSS_plus*, `Conclusion`.

7.1.2 General Results

As major results, we'd like to emphasize:

- A common Modelling Framework giving Semantics to MOSAR and Foretellix's M-SDL
- ... which can handle all known scenarios such as *single-lane*, *cut*, *crossing*, while maintaining openness for new actor-models
- precise meanings for, e. g., concepts such as *driving strategies*, *jerk*, *comfort*, and *driving objectives*, ...
- precise meanings for, e. g., concepts such as *scenario-classes*
- proofs on abstract relations between driving strategies (allowing arguments such as "$drive_{safe}$ is stronger as $motion_{rss}$ but offers more comfort and practicability")
- proofs on **safety** of three driving strategies
- by-products of proofs :

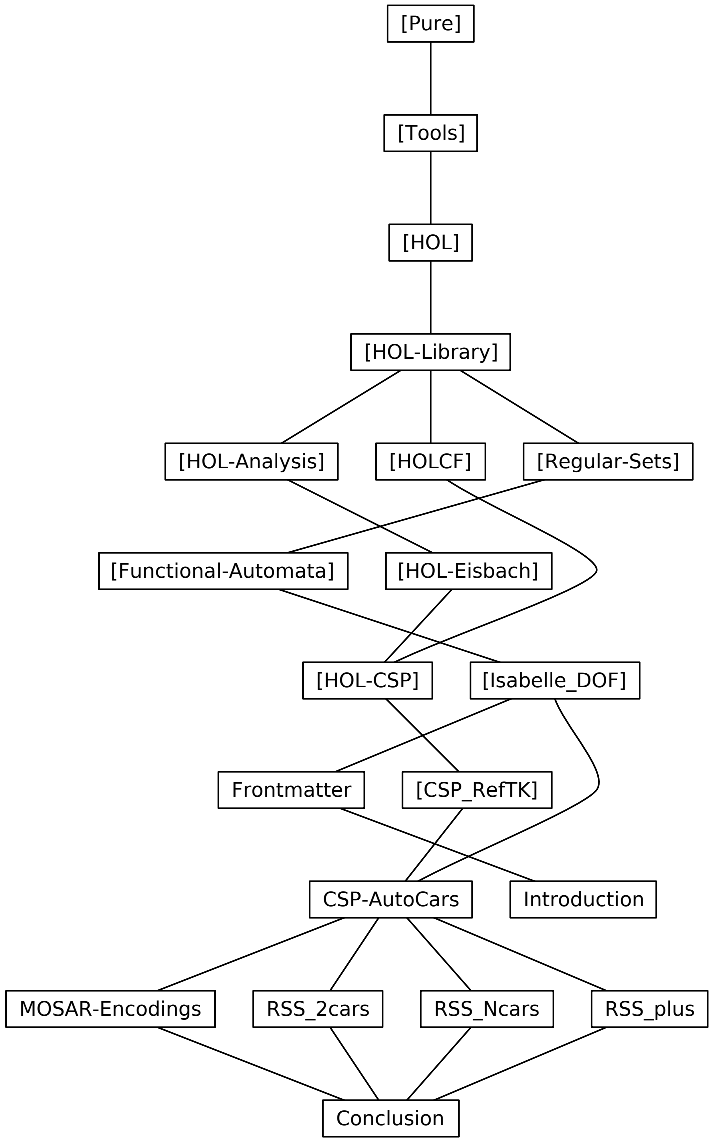


Figure 7.1: The Graph of Library-, Background-, and Foreground Theories of this Study

1. a precise understanding of the discretization effects of the continuous, cyber-physical models as occurring in simulators and concrete discrete controllers,
2. offer completeness for partitionings of *scenario-classes*, and therefore more trust than just simulations, and
3. proofs offered insight into behaviour and gave inspirations to unexpected improvements.

- HOL-CSP provided reasonable analytic power by semi-automated proof in order to formally prove key safety properties for a relevant driving strategy intensively discussed in the literature. The work comprises 400 loc for RSS, 450 loc for a generalisation RSS-N and an substantial improvement, RSS+ with 570 loc.
- The proof work alone comprises about 3 man months of work; future automation may bring down this effort considerably.
- the proofs give rise to a case-distinction structure that result in abstract test-cases that can be used for system tests within a certification (see Table 7.2,Table 7.3, and Table 7.4).
- the proofs reveal the completeness of the case-distinction underlying the abstract test cases which couldn't be established otherwise.
- a run to generate this document (which includes: type-checking, proof-checking, ontological consistency-checking, LaTeX generation and .pdf generation) takes 55 secs elapsed time on a common machine (MacBook Pro, 2,9 Ghz, 6 Core).

7.1.3 Abstract Test-Cases derived from the Proof-Structure

Our proof is based on induction. A simulation step δt defines the induction step. We can produce test cases following the case-splits we used in our proof. Therefore, our test cases are one step tests but we can randomly complete them or even combine them to obtain many-steps test cases. We will first give test cases for the standard RSS driving strategy drive_{rss}, and then we will consider test cases for our new driving strategy drive_{safe}. Requiring a smaller safety distance the safety proof of this latter requires more distinctions and will give more test cases.

Abstract Test-Cases for driving strategy drive_{rss}

Extracting the structure of case-distinctions from our proof, there are two testing objectives:

1. the minimal distance is preserved $d_{min} \equiv b_r - b_f$ that is the difference between the rear-car breaking distance $b_r = (\mathit{speed}\ (\omega\ i))^2\ /\ ((2::'a) * a_{minbrake})$ and the front car breaking distance $b_f = (\mathit{speed}\ (\omega\ (i + (1::'c))))^2\ /\ ((2::'a) * a_{maxbrake})$

2. no collision between the two cars when the previous distance is negative (i.e. $b_r < b_f$)

Minimal distance test cases: all cases are exclusive.

No	Test Case description	Conds
1	r-car stops safe distance f-car stops	$speed(\omega'\ i) = 0$ $d_{rss}\ \omega\ i \leq d_{real}\ \omega\ i$ $speed(\omega'\ (i{+}1)) = 0$
2	r-car stops safe distance f-car doesn't stop (breaking or accelerating)	$speed(\omega'\ i) = 0$ $d_{rss}\ \omega\ i \leq d_{real}\ \omega\ i$ $speed(\omega'\ (i{+}1)) > 0$
3	r-car doesn't stop (breaking or accelerating) safe distance f-car stops	$speed(\omega'\ i) > 0$ $d_{rss}\ \omega\ i \leq d_{real}\ \omega\ i$ $speed(\omega'\ (i{+}1)) = 0$
4	r-car doesn't stop (breaking or accelerating) safe distance f-car doesn't stop (breaking or accelerating)	$speed(\omega'\ i) > 0$ $d_{rss}\ \omega\ i \leq d_{real}\ \omega\ i$ $speed(\omega'\ (i{+}1)) > 0$
5	r-car stops non-safe distance minimal distance f-car stops	$speed(\omega'\ i) = 0$ $d_{rss}\ \omega\ i > d_{real}\ \omega\ i$ $d_{min}\ \omega\ i \leq d_{real}\ \omega\ i$ $speed(\omega'\ (i{+}1)) = 0$
6	r-car stops non-safe distance non-minimal distance f-car stops	$speed(\omega'\ i) = 0$ $d_{rss}\ \omega\ i > d_{real}\ \omega\ i$ $d_{min}\ \omega\ i > d_{real}\ \omega\ i$ $speed(\omega'\ (i{+}1)) = 0$
7	r-car stops non-safe distance minimal distance f-car doesn't stop (breaking or accelerating)	$speed(\omega'\ i) = 0$ $d_{rss}\ \omega\ i > d_{real}\ \omega\ i$ $d_{min}\ \omega\ i \leq d_{real}\ \omega\ i$ $speed(\omega'\ (i{+}1)) > 0$
8	r-car stops non-safe distance non-minimal distance f-car doesn't stop (breaking or accelerating)	$speed(\omega'\ i) = 0$ $d_{rss}\ \omega\ i > d_{real}\ \omega\ i$ $d_{min}\ \omega\ i > d_{real}\ \omega\ i$ $speed(\omega'\ (i{+}1)) > 0$
9	r-car doesn't stop (breaking or accelerating) non-safe distance minimal distance f-car stops	$speed(\omega'\ i) > 0$ $d_{rss}\ \omega\ i > d_{real}\ \omega\ i$ $d_{min}\ \omega\ i \leq d_{real}\ \omega\ i$ $speed(\omega'\ (i{+}1)) = 0$
10	r-car doesn't stop (breaking or accelerating) non-safe distance non-minimal distance f-car stops	$speed(\omega'\ i) > 0$ $d_{rss}\ \omega\ i > d_{real}\ \omega\ i$ $d_{min}\ \omega\ i > d_{real}\ \omega\ i$ $speed(\omega'\ (i{+}1)) = 0$
11	r-car doesn't stop (breaking or accelerating) non-safe distance minimal distance f-car doesn't stop (breaking or accelerating)	$speed(\omega'\ i) > 0$ $d_{rss}\ \omega\ i > d_{real}\ \omega\ i$ $d_{min}\ \omega\ i \leq d_{real}\ \omega\ i$ $speed(\omega'\ (i{+}1)) > 0$
12	r-car doesn't stop (breaking or accelerating) non-safe distance non-minimal distance f-car doesn't stop (breaking or accelerating)	$speed(\omega'\ i) > 0$ $d_{rss}\ \omega\ i > d_{real}\ \omega\ i$ $d_{min}\ \omega\ i > d_{real}\ \omega\ i$ $speed(\omega'\ (i{+}1)) > 0$

Table 7.1: Table with Abstract Test Cases for minimal distance of $drive_{rss}$

No collision: cases are not exclusive (see Table 7.2.)

Abstract Test-Cases for RSS++ version *drive*$_{safe}$

Following our proof, there are again two testing objectives: minimal distance and no-collision. In the following and to avoid a long enumeration, we do not specify the behavior of the front car for many abstract test cases. The reader can derive specific test cases by selecting any behavior for the front car (e.g. breaking, slowing, accelerating, stopping...)

Minimal distance test cases: (see Table 7.3.).

No collision: (see Table 7.4.)

7.2 Lessons Learnt

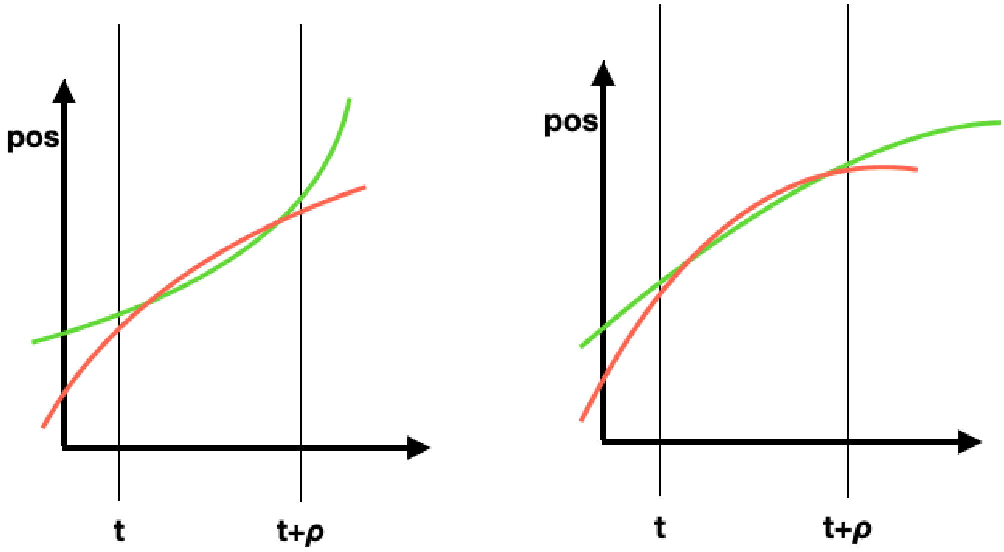
- The present study confirms that HOL-CSP is an adequate framework to give semantics to MOSAR- and similar models as used in the Automotive Car industry.
- More generally speaking, Isabelle/HOL-CSP — as being parametric for arbitrary HOL-types for events — is a suitable framework for modeling and analysing scenarios of cyber-physical systems involving both physical environments, communication, and discretisation problems of controllers.
- It provides distinctively more expressive power than conventional model- checking techniques such as, e. g., or PRISM [5], Hybrid CSP ([6], never implemented) or Hybrid Automata (numerous publications by Henzinger et al, but no usable tools).

Concerning the formal analysis of RSS, our study revealed a number of shortcomings in the existing description and analysis in [9].

We distinguish two lines of criticism: A *local critique* concerning modeling details of RSS and its safety proof, and a *global critique* on RSS concerning applicability and and the possible generalizations that the authors claim.

Our analysis revealed the following local problems, mostly due to the implicit assumption that all relevant functions are linear between t and $t+\varrho$:

- Case not treated in paper-and-pencil proof: if b-car behind, f-car in front, and if b is braking, and if f is accelerating, and if b-car behind and f-car in front at $t+\varrho$ there will be no collision. Counter-example: Figure 7.2 (left).
- Case not treated in paper-and-pencil proof: if b-car behind, f-car in front, and if b is braking, and if f is braking, and if b-car behind and f-car in front at $t+\varrho$ there will be no collision. Counter-example: Figure 7.2 (right).
- Case insufficiently treated: inflexion point (singularity) for $v_r = 0$, e. g. Figure 7.3 (left).
- General solution: distinguishing ϱ (the "reaction time" of the autonomous vehicle) from the sampling rate δt which must be small compared to ϱ in a simulator and infinitesimal in the proof (see Figure 7.3 (right)).



(a) b-car braking, f-car accelerating.

(b) b-car braking, f-car braking.

Figure 7.2: Collision scenarios overlooked.

Both phenomena — singularities as well as the need for an explicit sampling allowing for the construction of majorands and minorands — complicate the argument drastically and bring it to the limit of precision in usual paper-and-pencil proofs. However, we believe that the introduction of the sampling is inherently necessary for refinements towards concrete implementations.

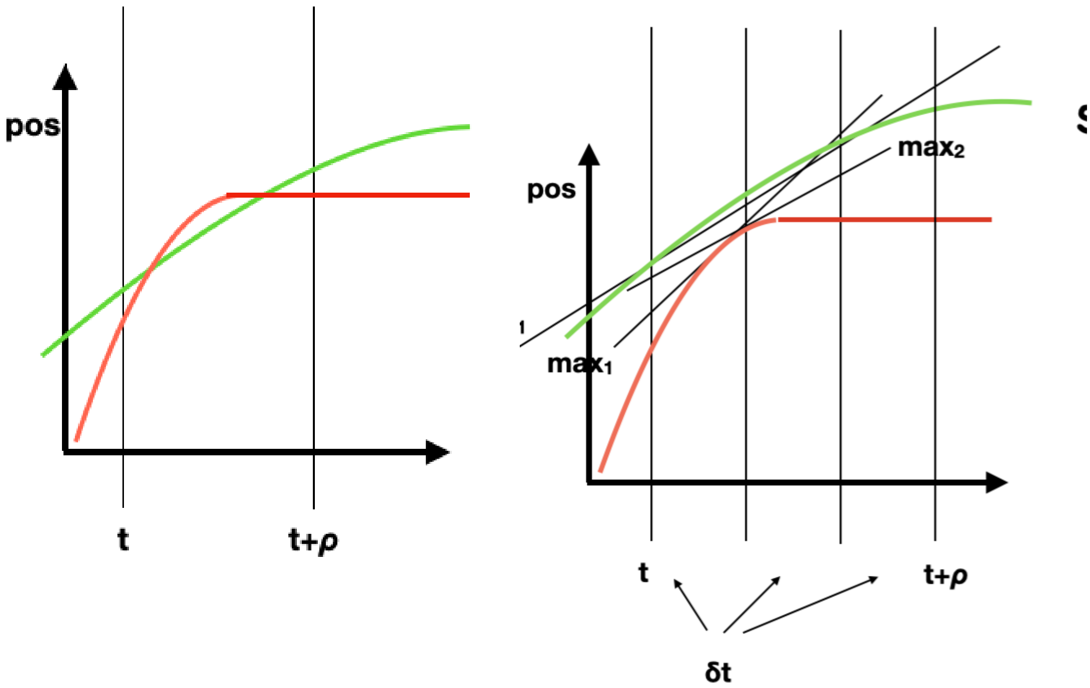
Concerning the global points of critique:

- the distances in a realistic configuration of RSS (speeds 133 km/h, ϱ=*0,1s*) results in nearly 60 m minimal distance ... (by courtesy of Paolo Crisafulli).
- given the large security distances, the argument of the RSS authors: "RSS can also handle curved street topologies and obstacles" is doubtful.

We believe that distances to side-walks need an own treatment in the driving strategy. More refined driving strategies will be out of reach of paper-and-pencil proofs and require a formal approach as the one discussed.

7.3 Future Directions

- Compilers from Foretellix's M-SDL or MOSAR to our Framework



(a) hand-waving: singuliarities.

(b) Solution: Explicit, infinitesimal sampling.

Figure 7.3: Phenomena overlooked.

- Simulators generated from sufficiently complete scenario instances
- Proofs on RSS are still one-dimensional (but an extension to 2-dimensional case is straight-forward similar to the original paper).
- More driving strategies taking into account jerk (and therefore comfort of the passengers)
- Going beyond safety properties towards lifeness properties (can the objective of the journey still be reached ?)
- driving strategies in our model are still functions in the continuous real-time-space underlying Newtonian physics. Isabelle/HOL and HOL-CSP allow for modeling and refinement proofs towards concrete test-diver implementation in, for example, C, modeling concrete calculations on machine-types such as bit-vectors.

No	Test Case description	Conds
1	r-car stops	$speed(\omega'\ i) = 0$
	f-car breaking faster	$\langle b_f < b_r \rangle$
	f-car stops	$speed(\omega'\ (i+1)) = 0$
2	r-car doesn't stop (breaking or accelerating)	$speed(\omega'\ i) > 0$
	f-car breaking faster	$\langle b_f < b_r \rangle$
	f-car stops	$speed(\omega'\ (i+1)) = 0$
3	r-car stops	$speed(\omega'\ i) = 0$
	f-car breaking faster	$\langle b_f < b_r \rangle$
	non-safe distance	$d_{rss}\ \omega\ i > d_{real}\ \omega\ i$
	minimal distance	$d_{min}\ \omega\ i \leq d_{real}\ \omega\ i$
	f-car doesn't stop (breaking or accelerating)	$speed(\omega'\ (i+1)) > 0$
	f-car very slow (could stop)	$speed(\omega\ (i+1)) - a_{maxbrake} * \delta t \leq 0$
4	r-car doesn't stop (breaking or accelerating)	$speed(\omega'\ i) > 0$
	f-car breaking faster	$\langle b_f < b_r \rangle$
	non-safe distance	$d_{rss}\ \omega\ i > d_{real}\ \omega\ i$
	minimal distance	$d_{min}\ \omega\ i \leq d_{real}\ \omega\ i$
	f-car doesn't stop (breaking or accelerating)	$speed(\omega'\ (i+1)) > 0$
	f-car very slow (could stop)	$speed(\omega\ (i+1)) - a_{maxbrake} * \delta t \leq 0$
5	r-car stops	$speed(\omega'\ i) = 0$
	f-car breaking faster	$\langle b_f < b_r \rangle$
	non-safe distance	$d_{rss}\ \omega\ i > d_{real}\ \omega\ i$
	non-minimal distance	$d_{min}\ \omega\ i > d_{real}\ \omega\ i$
	f-car doesn't stop (breaking or accelerating)	$speed(\omega'\ (i+1)) > 0$
	f-car very slow (could stop)	$speed(\omega\ (i+1)) - a_{maxbrake} * \delta t \leq 0$
6	r-car doesn't stop (breaking or accelerating)	$speed(\omega'\ i) > 0$
	f-car breaking faster	$\langle b_f < b_r \rangle$
	non-safe distance	$d_{rss}\ \omega\ i > d_{real}\ \omega\ i$
	non-minimal distance	$d_{min}\ \omega\ i > d_{real}\ \omega\ i$
	f-car doesn't stop (breaking or accelerating)	$speed(\omega'\ (i+1)) > 0$
	f-car very slow (could stop)	$speed(\omega\ (i+1)) - a_{maxbrake} * \delta t \leq 0$
ls 7	r-car stops	$speed(\omega'\ i) = 0$
	r-car breaking faster	$\langle b_f \geq b_r \rangle$
	f-car stops	$speed(\omega'\ (i+1)) = 0$
8	r-car doesn't stop (breaking or accelerating)	$speed(\omega'\ i) > 0$
	r-car breaking faster	$\langle b_f \geq b_r \rangle$
	f-car stops	$speed(\omega'\ (i+1)) = 0$
9	r-car stops	$speed(\omega'\ i) = 0$
	r-car breaking faster	$\langle b_f \geq b_r \rangle$
	f-car doesn't stop (breaking or accelerating)	$speed(\omega'\ (i+1)) > 0$
10	r-car doesn't stop (breaking or accelerating)	$speed(\omega'\ i) > 0$
	r-car breaking faster	$\langle b_f \geq b_r \rangle$
	f-car doesn't stop (breaking or accelerating)	$speed(\omega'\ (i+1)) > 0$
11	r-car stops	$speed(\omega'\ i) = 0$
	f-car breaking faster	$\langle b_f < b_r \rangle$
	non-safe distance	$d_{rss}\ \omega\ i > d_{real}\ \omega\ i$
	minimal distance	$d_{min}\ \omega\ i \leq d_{real}\ \omega\ i$
	f-car doesn't stop (breaking or accelerating)	$speed(\omega'\ (i+1)) > 0$
	f-car not very slow (could not stop)	$speed(\omega\ (i+1)) - a_{maxbrake} * \delta t > 0$
12	r-car doesn't stop (breaking or accelerating)	$speed(\omega'\ i) > 0$
	f-car breaking faster	$\langle b_f < b_r \rangle$
	non-safe distance	$d_{rss}\ \omega\ i > d_{real}\ \omega\ i$
	minimal distance	$d_{min}\ \omega\ i \leq d_{real}\ \omega\ i$
	f-car doesn't stop (breaking or accelerating)	$speed(\omega'\ (i+1)) > 0$
	f-car not very slow (could not stop)	$speed(\omega\ (i+1)) - a_{maxbrake} * \delta t > 0$
13	r-car stops	$speed(\omega'\ i) = 0$
	f-car breaking faster	$\langle b_f < b_r \rangle$
	non-safe distance	$d_{rss}\ \omega\ i > d_{real}\ \omega\ i$
	non-minimal distance	$d_{min}\ \omega\ i > d_{real}\ \omega\ i$
	f-car doesn't stop (breaking or accelerating)	$speed(\omega'\ (i+1)) > 0$
	f-car not very slow (could not stop)	$speed(\omega\ (i+1)) - a_{maxbrake} * \delta t > 0$
14	r-car doesn't stop (breaking or accelerating)	$speed(\omega'\ i) > 0$
	f-car breaking faster	$\langle b_f < b_r \rangle$
	non-safe distance	$d_{rss}\ \omega\ i > d_{real}\ \omega\ i$
	non-minimal distance	$d_{min}\ \omega\ i > d_{real}\ \omega\ i$
	f-car doesn't stop (breaking or accelerating)	$speed(\omega'\ (i+1)) > 0$
	f-car not very slow (could not stop)	$speed(\omega\ (i+1)) - a_{maxbrake} * \delta t > 0$

Table 7.2: Table with Abstract Test Cases for no-collision of$drive_{rss}$

No	Test Case description	Conds
1	r-car stops	$speed(\omega'\ i) = 0$
	safe distance	$d_{safe}\ \omega\ i\ acc(\omega'\ i) + \varepsilon' \leq d_{real}\ \omega\ i$
	f-car stops	$speed(\omega'\ (i+1)) = 0$
2	r-car stops	$speed(\omega'\ i) = 0$
	safe distance	$d_{safe}\ \omega\ i\ acc(\omega'\ i) + \varepsilon' \leq d_{real}\ \omega\ i$
	f-car doesn't stop (breaking or accelerating)	$speed(\omega'\ (i+1)) > 0$
3	r-car doesn't stop (breaking or accelerating)	$speed(\omega'\ i) > 0$
	safe distance	$d_{safe}\ \omega\ i\ acc(\omega'\ i) + \varepsilon' \leq d_{real}\ \omega\ i$
	f-car stops	$speed(\omega'\ (i+1)) = 0$
4	r-car doesn't stop (breaking or accelerating)	$speed(\omega'\ i) > 0$
	safe distance	$d_{safe}\ \omega\ i\ acc(\omega'\ i) + \varepsilon' \leq d_{real}\ \omega\ i$
	f-car doesn't stop (breaking or accelerating)	$speed(\omega'\ (i+1)) > 0$
5	r-car stops	$speed(\omega'\ i) = 0$
	r-car was breaking	$acc(\omega'\ i) \leq a_{minbrake},$
	safe distance	$d_{safe}\ \omega\ i\ acc(\omega'\ i) + \varepsilon' \leq d_{real}\ \omega\ i$
	f-car stops	$speed(\omega'\ (i+1)) = 0$
6	r-car stops	$speed(\omega'\ i) = 0$
	r-car was breaking	$acc(\omega'\ i) \leq a_{minbrake},$
	safe distance	$d_{safe}\ \omega\ i\ acc(\omega'\ i) + \varepsilon' \leq d_{real}\ \omega\ i$
	f-car doesn't stop (breaking or accelerating)	$speed(\omega'\ (i+1)) > 0$
7	r-car stops	$speed(\omega'\ i) = 0$
	r-car was not breaking (slowing)	$acc(\omega'\ i) > a_{minbrake},$
	safe distance	$d_{safe}\ \omega\ i\ acc(\omega'\ i) + \varepsilon' \leq d_{real}\ \omega\ i$
	f-car stops	$speed(\omega'\ (i+1)) = 0$
8	r-car stops	$speed(\omega'\ i) = 0$
	r-car was not breaking but slowing	$acc(\omega'\ i) > a_{minbrake},$
	safe distance	$d_{safe}\ \omega\ i\ acc(\omega'\ i) + \varepsilon' \leq d_{real}\ \omega\ i$
	f-car doesn't stop (breaking or accelerating)	$speed(\omega'\ (i+1)) > 0$
9	r-car doesn't stop	$speed(\omega'\ i) > 0$
	r-car was breaking	$acc(\omega'\ i) \leq a_{minbrake},$
	safe distance	$d_{safe}\ \omega\ i\ acc(\omega'\ i) + \varepsilon' \leq d_{real}\ \omega\ i$
10	r-car doesn't stop (breaking or accelerating)	$speed(\omega'\ i) > 0$
	r-car accelerates	$acc(\omega'\ i) \geq 0$
	safe distance	$d_{safe}\ \omega\ i\ acc(\omega'\ i) + \varepsilon' \leq d_{real}\ \omega\ i$
11	r-car doesn't stop	$speed(\omega'\ i) > 0$
	r-car was not breaking but slowing	$a_{minbrake} < acc(\omega'\ i) \leq 0$
	r-car would stop after response time	$speed(\omega\ i) +\ ?a_0 * \varrho \leq 0$
	safe distance	$d_{safe}\ \omega\ i\ acc(\omega'\ i) + \varepsilon' \leq d_{real}\ \omega\ i$
12	r-car doesn't stop	$speed(\omega'\ i) > 0$
	r-car was not breaking but slowing	$a_{minbrake} < acc(\omega'\ i) \leq 0$
	r-car would not stop after response time	$speed(\omega\ i) +\ ?a_0 * \varrho > 0$
	safe distance	$d_{safe}\ \omega\ i\ acc(\omega'\ i) + \varepsilon' \leq d_{real}\ \omega\ i$

Table 7.3: Table with Abstract Test Cases for minimal distance of $drive_{safe}$ (RSS++)

No	Test Case description	Conds
1	r-car stops	$speed(\omega'\ i) = 0$
	f-car breaking faster	$\langle b_f < b_r \rangle$
	f-car stops	$speed(\omega'\ (i+1)) = 0$
2	r-car doesn't stop (breaking or accelerating)	$speed(\omega'\ i) > 0$
	f-car breaking faster	$\langle b_f < b_r \rangle$
	f-car stops	$speed(\omega'\ (i+1)) = 0$
3	f-car breaking faster	$\langle b_f < b_r \rangle$
	safe distance	$d_{safe}\ \omega\ i \leq d_{real}\ \omega\ i$
	f-car doesn't stop (breaking or accelerating)	$speed(\omega'\ (i+1)) > 0$
	f-car very slow (could stop)	$speed(\omega\ (i+1)) + a_{maxbrake} * \delta t \leq 0$
4	f-car breaking faster	$\langle b_f < b_r \rangle$
	non-safe distance	$d_{safe}\ \omega\ i > d_{real}\ \omega\ i$
	minimal distance	$d_{min}\ \omega\ i \leq d_{real}\ \omega\ i$
	f-car doesn't stop (breaking or accelerating)	$speed(\omega'\ (i+1)) > 0$
	f-car very slow (could stop)	$speed(\omega\ (i+1)) - a_{maxbrake} * \delta t \leq 0$
5	r-car stops	$speed(\omega'\ i) = 0$
	f-car breaking faster	$\langle b_f < b_r \rangle$
	f-car doesn't stop	$speed(\omega'\ (i+1)) > 0$
	f-car not very slow (could not stop)	$speed(\omega\ (i+1)) - a_{maxbrake} * \delta t > 0$
6	r-car doesn't stop	$speed(\omega'\ i) > 0$
	r-car was breaking	$acc(\omega'\ i) \leq a_{minbrake}$
	f-car breaking faster	$\langle b_f < b_r \rangle$
	f-car doesn't stop	$speed(\omega'\ (i+1)) > 0$
	f-car not very slow (could not stop)	$speed(\omega\ (i+1)) - a_{maxbrake} * \delta t > 0$
7	r-car doesn't stop	$speed(\omega'\ i) > 0$
	r-car was not breaking (slowing or accelerating)	$acc(\omega'\ i) > a_{minbrake}$
	f-car breaking faster	$\langle b_f < b_r \rangle$
	f-car doesn't stop	$speed(\omega'\ (i+1)) > 0$
	f-car not very slow (could not stop)	$speed(\omega\ (i+1)) - a_{maxbrake} * \delta t > 0$

Table 7.4: Table with Abstract Test Cases for no-collision of$drive_{safe}$ (RSS++)

Bibliography

[1] T. C. I. 22 and S. S. 32. Road vehicles — safety of the intended functionality. techreport iso/pas 21448:2019, International Organization for Standardization, 2019.

[2] S. D. Brookes and A. W. Roscoe. An improved failures model for communicating sequential processes. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, pages 281–305, Berlin, Heidelberg, 1985. Springer. ISBN 978-3-540-39593-5.

[3] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[4] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985. ISBN 0-13-153271-5.

[5] M. Kwiatkowska, G. Norman, D. Parker, and G. Santos. Prism-games 3.0: Stochastic game verification with concurrency, equilibria and time. In S. K. Lahiri and C. Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, volume 12225 of *Lecture Notes in Computer Science*, pages 475–487. Springer, 2020. doi: 10.1007/978-3-030-53291-8_25. URL https://doi.org/10.1007/978-3-030-53291-8_25.

[6] J. Liu, J. Lv, Z. Quan, N. Zhan, H. Zhao, C. Zhou, and L. Zou. A calculus for hybrid CSP. In K. Ueda, editor, *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings*, volume 6461 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2010. doi: 10.1007/978-3-642-17164-2_1. URL https://doi.org/10.1007/978-3-642-17164-2_1.

[7] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. doi: 10.1007/3-540-45949-9.

[8] A. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall, 1997. ISBN 0-13-674409-5.

[9] S. Shalev-Shwartz, S. Shammah, and A. Shashua. On a Formal Model of Safe and Scalable Self-driving Cars. *arXiv e-prints*, art. arXiv:1708.06374, Aug. 2017.

[10] S. Taha, B. Wolff, and L. Ye. Philosophers may dine - definitively! In B. Dongol and E. Troubitsyna, editors, *Integrated Formal Methods - 16th International Conference, IFM 2020, Lugano, Switzerland, November 16-20, 2020, Proceedings*, volume 12546 of *Lecture Notes in Computer Science*, pages 419–439. Springer, 2020. doi: 10.1007/978-3-030-63461-2_23. URL https://doi.org/10.1007/978-3-030-63461-2_23.