Formal Verification of Automated Driving: RSS and Safety Architectures

Clovis Eberhart, Jérémy Dubut, James Haydon, Ichiro Hasuo

We present our work on Responsibility-Sensitive Safety (RSS). RSS is a methodology to mathematically prove safety of automated driving. We formalise driving scenarios as programs containing differential equations. We develop dFHL, a logic based on Hoare logic, to prove properties of such programs. This allows us to prove properties of complex scenarios. Safety architectures, and in particular the simplex architecture, play a crucial role in safety of automated driving. We extend dFHL with proof rules tailored to prove properties of safety architectures. This allows us to formally define of the simplex architecture and formally prove its safety, as well as define new architectures and prove their safety.

1 Introduction

Automated driving has been a seemingly achievable dream for more than a decade now. The underlying benefits would be a reduced number of accidents, more efficient transportation systems, and a positive environmental impact by changing the transportation paradigm from a personal vehicles to shared mobility, to name only a few [1]. There is an impressive body of work on automated driving, stemming from different communities such as control theory, communication systems, artificial intelligence and machine learning.

While there has been public interest for a few years around 2018 following impressive progress by car manufacturers, it seems that it has died down in recent years after some deadly crashes involving automated driving systems. There are of course existing approaches to safety for automated driving, but they focus on statistical safety. These approaches usually consist in either testing [6] or collecting statistics on automated vehicles on real roads. However, the guarantees offered by these approaches are unclear, and their explainability is limited. On the other hand, we pursue mathematical proofs of safety, which offer both a strong guarantee (as long as the hypotheses hold, the conclusion of theorem holds unconditionally) and explainability (the proof itself is a detailed explanation). One such approach is runtime verification for automated driving [5] [7], but these cannot prove the safety of a manoeuvre for a variety of instances, only for the particular instance it is facing.

Responsibility-Sensitive Safety

Responsibility-Sensitive Safety (RSS) [10] is an existing approach that falls in the category of math-

---

* 自動運転の形式的検証　ＲＳＳと安全アーキテクチャ
The work is partially supported by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603) and ACT-I (No. JPMJPR17UA), JST; and Grants-in-aid No. 19K20215 & 19K20249, JSPS.

Clovis Eberhart, 国立情報学研究所, National Institute of Informatics.
Jérémy Dubut, 国立研究開発法人産業技術総合研究所, National Institute of Advanced Industrial Science and Technology.
James Haydon, 国立情報学研究所, National Institute of Informatics.
Ichiro Hasuo, 国立情報学研究所, National Institute of Informatics.

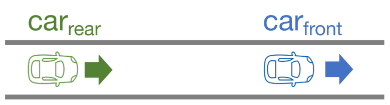

図 1　**Single-lane follow**

ematical proofs of safety for automated driving. For a given situations, the RSS method gives: 1) a mathematical formula called the *RSS condition* and 2) a control strategy called the *proper response*. The idea is that, if a vehicle is in a state such that the RSS condition holds and it executes the proper response, then it is not responsible for any collisions. In particular, if all vehicles apply this behaviour, then there are no collisions (under the assumptions that there is at least one agent responsible in any collision).

Example 1. *To understand RSS better, let us look at the archetypal example of RSS, called the* single-lane follow *scenario, which is depicted in Figure 1. There, the Subject Vehicle (*SV*) is the following car and the Principal Other Vehicle (*POV*) is the leading car. Since* SV *is behind* POV *and they are moving in the same direction,* SV *is deemed responsible for any collision. Therefore, we need to find an RSS condition and a proper response under which* SV *avoids all collisions.*

We make the following assumptions:

- SV *and* POV *behave like one-dimensional point masses, so their dynamic behaviour is*
$$\begin{cases} \dot{x} &= v \\ \dot{v} &= a, \end{cases}$$
where x, v, and a respectively represent their positions, velocities, and accelerations,
- POV *can only accelerate between rates $-b_{\max}$ (the maximum brake rate) and $a_{\max}$ (the maximum acceleration),*
- POV *never drives backwards (its velocity is non-negative),*
- SV *only reacts after time ρ (the* response time*) has elapsed,*
- *we want* SV *to only accelerate at rates between $-b_{\min}$ (the maximum comfortable brake rate) and $a_{\max}$.*

Then we can derive that the following proper response avoids all collisions when the RSS condition holds. Proper response: SV *brakes at rate $-b_{\min}$ as soon as it starts reacting. RSS condition: the distance between the two vehicles is greater than* $\mathsf{dRSS}(v_{\mathrm{f}}, v_{\mathrm{r}}, \rho)$*, which is defined as*

$$\max\left(0, v_{\mathrm{r}}\rho + \frac{a_{\max}\rho^2}{2} + \frac{(v_{\mathrm{r}} + a_{\max}\rho)^2}{2b_{\min}} - \frac{v_{\mathrm{f}}^2}{2b_{\max}}\right)$$

While it is possible to derive RSS conditions and proper responses for simple scenarios such as the one above, it becomes unfeasible for humans on more complex ones that involve multiple POVs and constraints. Our approach is thus to apply program logic to reason about driving scenarios. The advantages are that it is more scalable and less error-prone. Such an approach can thus help derive RSS conditions and proper responses for more complex scenarios, which was impossible to do by hand. We design a language to model driving scenarios and a logic dFHL to reason about them.

Safety Architectures in Automated Driving

Our goal is to create a methodology to design RSS conditions and proper responses for a wide variety of scenarios. However, such responses cannot usually be used on their own, either because they should only be used when safety is critical, or because they do not cover all possible driving situations. *Safety architectures* are techniques that allow a vehicle to run using one controller most of the time, then switch to another controller when some property holds.

The most commonly used safety architecture is the *simplex architecture* [2] [9], depicted in Figure 2. There, the Advanced Controller (AC) runs most of the time, but when the Decision Module (DM) deems that safety is critical, control is handed over to the Baseline Controller (BC), which focuses only

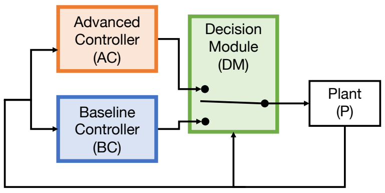

図 2　**The simplex architecture**

on safety.

RSS can easily be implemented using a simplex architecture:

- *AC* is an advanced controller that is optimised for many parameters (safety, comfort, fuel efficiency, etc.), and which does not have any formal guarantees,
- *BC* executes the proper response,
- *DM* hands the control to *BC* when the RSS condition is about to be violated.

We thus want to not only reason about *BC*, but also about the whole safety architecture comprising *BC* and other components. We thus extend the logic dFHL to dFHL$^{\downarrow}$, which has some built-in features to reason about safety architectures.

Contributions

This paper is an informal introduction to our previous papers [4] [3]. While it contains definitions and lemmas, not all details are provided (for example, only one derivation rule of dFHL is shown) to give a flavour of what the logic is about and how it is used. The reader is invited to read the original papers for more details.

Overview

In Section 2, we define the syntax and operational semantics of the language that we use to model automated driving scenarios, as well as the formalism that we use to specify properties of such programs. In Section 3, we present the logic dFHL, the Hoare-style logic that we use to derive properties of programs. This logic gets extended to dFHL$^{\downarrow}$ in Section 4, where this extension allows us to reason about safety architectures.

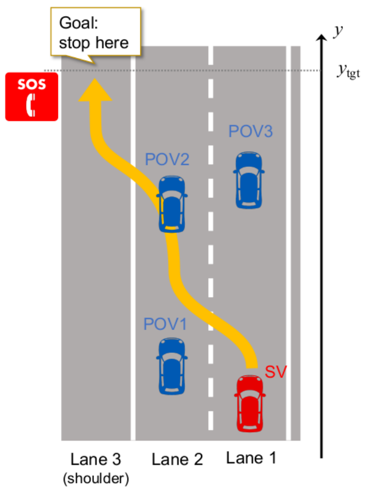

図 3　**The emergency pull over scenario**

about safety architectures.

2 Hybrid Programs: Syntax and Semantics

In this section, we present the language of hybrid programs that are used to model driving scenarios.

2.1 Driving Scenarios

We first give an informal idea of what we mean by “driving scenario”. Figure 3, represents the pull over scenario, in which the Subject Vehicle (SV) must reach a given goal with zero velocity while avoiding all Principle Other Vehicles (POVs).

A *driving scenario* consists of:

- a given topology of the road (here, a straight road with three lanes),
- relative positions of POVs (here, POV3 in Lane 1 in front of SV, POV1 in Lane 2, and POV2 in Lane 2 in front of POV1),
- and behaviours for POVs (here, we assume that all POVs stay in their respective lanes, and they drive with constant speed bound by some known constants $v_{\min}$ and $v_{\max}$, unless another vehicle is in front of them with less than an RSS safety distance, in which case they brake until

the RSS safety distance is restored),

- a goal for SV (here, stopping at a designated point on Lane 3),
- a safety condition for SV (here, avoiding collisions with all POVs).

A driving scenario can give rise to many *instances*, where the topological constraints are instantiated with real values on variables. Here for example, each different tuple of values of the positions of the POVs, the speeds of all vehicles, and the distance to the target area give a different instance of the pull over driving scenario.

2.2 Syntax

Now, we want to define a language of programs that can represent such driving scenarios (or their instances). This language needs to contain general programming constructions to model discrete control (e.g., the different choices that vehicles can make), but also constructs that represent the continuous evolution of systems. We thus use a model of hybrid programs inspired by differential dynamic logic [8].

Definition 2 (terms, assertions)**.** *A* term *is a rational polynomial on a fixed infinite set V of variables.* Assertions *are generated by the grammar*

$$A, B ::= \mathsf{true} \mid e \sim f \mid \neg A \mid A \wedge B,$$

where e, f are terms and $\sim\, \in \{=, \leq, <, \neq\}$.

As usual, we can derive all Boolean connectors by encoding them as $A \vee B \equiv \neg(\neg A \wedge \neg B)$, $A \Rightarrow B \equiv \neg A \vee B$, and $\mathsf{false} \equiv \neg\mathsf{true}$.

Definition 3 (programs)**.** *We assume that the set of variables $V = V_C \sqcup V_P \sqcup V_E$ is the disjoint union of V_C (*cyber variables*), V_P (*physical variables*), and V_E (*environment variables*).*

Hybrid program *(or* programs*) are generated by the grammar*

$$\alpha, \beta ::= \mathsf{skip} \mid x := e \mid \alpha; \beta \mid \mathsf{if}\,(A)\,\{\alpha\}\,\mathsf{else}\,\{\beta\} \mid$$
$$\mathsf{dwhile}\,(A)\,\{\dot{\mathbf{x}} = \mathbf{f}\} \mid \mathsf{while}\,(A)\,\{\alpha\}$$

In $x := e$, x is a cyber variable. In $\mathsf{dwhile}\,(A)\,\{\dot{\mathbf{x}} = \mathbf{f}\}$, $\mathbf{x}$ and $\mathbf{f}$ are lists of the same length, respectively of (distinct) (non-environment) variables and terms. We abbreviate $\mathsf{if}\,(A)\,\{\alpha\}\,\mathsf{else}\,\{\mathsf{skip}\}$ as $\mathsf{if}\,(A)\,\{\alpha\}$.

The basic idea is that:

- physical variables describe physical quantities, which can only evolve continuously (according to differential equations in our modelling),
- cyber variables are used by the logical controller, and can change discretely, as well as continuously,
- environment variables are variables that may change either discretely or continuously, but the program has no impact on them, they are changed independently from how the program reduces (a possible way to think about this is to think that the program is running in parallel with an unspecified environment program that can change these variables).

2.3 Semantics

We define an operational semantics for our language of hybrid programs. Since we are interested in safety properties such as absence of collisions and in termination, it is not enough to map states to end states. There are several possible ways to track the intermediate states of computations, and here we choose an LTL-style approach with explicit traces.

Definition 4 (store)**.** *A* store *is a function $\rho \colon V \to \mathbb{R}$ from variables to reals.* Store update *is denoted $\rho[x \to v]$; it maps x to v and any other variable x' to $\rho(x')$. The* value *$[\![e]\!]_\rho$ of a term e in a store ρ is a real defined as usual by induction on e. The* satisfaction *relation $\rho \models A$ between stores and assertions is also defined as usual. We write $\rho \sim \rho'$ when $\forall x \in V_C \sqcup V_P$, $\rho(x) = \rho'(x)$.*

In our approach, traces are tuples $(h_0, \ldots, h_n)$ of functions $h_i \colon [0, t_i] \to \mathbb{R}^V$. This allows us to account both for discrete steps by using several functions (on the interval $[0, 0]$) and continuous ones by using non-trivial intervals.

Definition 5 (trace)**.** *A* trace *is a (finite or infinite) sequence $\sigma = \big((t_0, h_0), (t_1, h_1), \ldots\big)$ of pairs, where $t_i \in \mathbb{R}_{\geq 0}$ and $h_i \colon [0, t_i] \to \mathbb{R}^V$ is a continuous function.*

If σ above is a sequence of length $n \in \overline{\mathbb{N}} = \mathbb{N} \cup \{+\infty\}$, we write $\mathrm{dom}(\sigma) \equiv \{(i,t) \,|\, i < n+1, t \leq t_i\}$, $\sigma(i) = h_i$ and $\sigma(i,t) = h_i(t)$ for $(i,t) \in \mathrm{dom}(\sigma)$. Given σ of finite length n, we define the ending state $\mathrm{end}(\sigma) \in \mathbb{R}^V$ *as $\sigma(n, t_n)$. Given an assertion C, we define $\sigma \vDash C$ as for all $(i,t) \in \mathrm{dom}(\sigma)$, $\sigma(i,t) \vDash C$. We denote by δ_ρ the trace $\big((0, f_\rho)\big)$, where $f_\rho(0) = \rho$. Given σ as above and $(i,t) \in \mathrm{dom}(\sigma)$, we define $\sigma_{|i,t} = \big((t_0, h_0), \ldots, (t_{i-1}, h_{i-1}), (t, h_{i|[0,t]})\big)$. The concatenation of a finite trace σ and a trace σ' is denoted $\sigma \cdot \sigma'$. Similarly, $\odot_{i=0}^{n}\sigma_i$ is the concatenation of traces σ_0, ..., σ_n, where all are finite (except maybe σ_n), and $\odot_{i=0}^{+\infty}\sigma_i$ the concatenation of finite σ_0, σ_1, ...*

We do not give the whole operational semantics of the language, only that of the dwhile construct, because it is the only non-standard construct of our language. Its semantics is also the most complex, since environment variables are allowed to change during its execution.

Definition 6 (operational semantics (dwhile only))**.** *The set of* valid *traces σ for a program α from a store ρ is defined by induction on α. The fact that σ is valid for α from ρ is denoted $\rho, \sigma \vDash \alpha$.*

For the dwhile *construct: $\rho, \sigma \vDash \mathsf{dwhile}\,(C)\,\{\dot{\mathbf{x}} = \mathbf{f}\}$ if and only if either $\rho \nvDash C$ and $\sigma = \delta_\rho$; or $\rho \vDash C$ and there exists $n \in \overline{\mathbb{N}}$ such that $\sigma = \big((t_i, h_i)\big)_{i<n}$, and*

- *for all $i < n$, $h_{i|\mathbf{x}}$ is differentiable, its derivative is $h'_{i|\mathbf{x}}(t) = [\![\mathbf{f}]\!]_{h_i(t)}$ for all $0 \leq t \leq t_i$, $h_i(x)$ is constant for other $x \in V_C \cup V_P$, and $h_i(0) \sim h_{i-1}(t_{i-1})$,*
- *for all $(i,t) \in \mathrm{dom}\,(\sigma)$, if $(i,t) < (n-1, t_{n-1})$, then $h_i(t) \vDash C$ (if $n \neq +\infty$),*
- *$h_{n-1}(t_{n-1}) \nvDash C$ (if $n \neq +\infty$),*

where $h_{-1}(t_{-1}) = \rho$, and $+\infty - 1 = +\infty$.

Informally, a trace for $\mathsf{dwhile}\,(A)\,\{\dot{\mathbf{x}} = \mathbf{f}\}$ is a tuple of functions $\big((t_i, h_i)\big)_{i<n}$, where:

- all program variables evolve continuously according to the differential equation $\dot{\mathbf{x}} = \mathbf{f}$,
- each h_i denotes a continuous part of the evolution of environment variables,
- a change from h_i to h_{i+1} describes a possible discrete jump in the values of the environment variables,
- the evolution continues until A becomes false, or continues forever otherwise.

While our language is non-deterministic because of the environment behaviour, it is actually deterministic once the behaviour of the environment has been fixed.

Example 7 (one-way traffic, modelling)**.** *In this scenario, two vehicles are in the situation described in Figure 1. We want to prove that* SV *can avoid collisions.*

In this scenario, x, x_{POV}, v, and v_{POV} are physical variables, and a_{POV} is an environment variable. The program of interest is α, which is defined as:

$$\begin{aligned}
\delta_{\mathtt{brake}} &\equiv \{\dot{x} = v, \dot{v} = -b_{\min},\\
&\qquad \dot{x}_{\mathsf{POV}} = v_{\mathsf{POV}}, \dot{v}_{\mathsf{POV}} = a_{\mathsf{POV}}\},\\
\delta_{\mathtt{accel}} &\equiv \{\dot{x} = v, \dot{v} = a_{\max},\\
&\qquad \dot{x}_{\mathsf{POV}} = v_{\mathsf{POV}}, \dot{v}_{\mathsf{POV}} = a_{\mathsf{POV}}\},\\
\alpha &\equiv \left[\begin{array}{l} t := 0; \mathsf{dwhile}\,(t < \rho)\,\{\delta_{\mathtt{accel}}, \dot{t} = 1\};\\ \mathsf{dwhile}\,(v > 0)\,\{\delta_{\mathtt{brake}}\} \end{array}\right].
\end{aligned}$$

It models the worst case for SV*'s behaviour, which is when* SV *accelerates until time ρ, then brakes at rate $-b_{\min}$, while* POV *can behave however they like.*

2.4 Hoare Quintuples

Now, we want to specify behaviours of programs. In order to do that, we use a formalism akin to Hoare triples. However, since we are not only interested in the input-output behaviour of the program, but also in the intermediate states, the precise formalism is slightly more complex than Hoare triples. We use *Hoare quintuples*, which are of the following form:

$$A : [P]\ \alpha\ [Q] : G.$$

The meaning is close to that of Hoare triples for total correctness: if program α runs from the *precondition* P, then it terminates in the *postcondition* Q. Moreover, all states during execution will satisfy the *guarantee* G. However, that is only under the condition that the *assumption* A holds throughout the execution.

There is one difficulty in satisfying termination in our context. Indeed, there are infinite traces that correspond to finite executions. For example, if the execution of a dwhile lasts for 3 seconds, then there exists a trace $\big((t_i, h_i)\big)_{i\in\mathbb{N}}$, where $t_i = 1/2^i$ (of total length of 2 seconds) cannot formally be extended, but it is still "incomplete" in the following sense.

Definition 8 (incomplete traces)**.** *An infinite trace of the form* $((t_i, h_i))_{i\in\mathbb{N}}$ *with all* $t_i > 0$ *induces a function* $h\colon [0, \sum_i t_i) \to \mathbb{R}^V$ *by for all* $\sum_{i\le n} t_i \le s < \sum_{i\le n+1}$*,* $h(s) = h_n(s - \sum_{i\le n} t_i)$*. We say that a valid trace for* α *from* ρ *of the form* $\sigma \cdot ((t_1, h_1), \dots, (t_n, h_n))$ *with* $\sum_i t_i < +\infty$ *is* incomplete *if there is a valid trace of the form* $\sigma \cdot (t', h') \cdot \sigma'$ *where* $t' \ge \sum_i t_i$ *and for all* $s < \sum_i t_i$*,* $h(s) = h'(s)$*.*

We can now formally state the correctness of Hoare quintuples.

Definition 9 (Hoare quintuple)**.** *A* Hoare quintuple $A : [P]\ \alpha\ [Q] : G$ *is* totally correct *(or simply* correct*) if, for all stores* $\rho \vDash P$ *and traces* $\sigma = \big((t_i, h_i)\big)_{i<n}$ *valid for* α *from* ρ*, then*

- termination*: if* $\sigma \vDash A$*, then* σ *is either finite or incomplete,*
- postcondition*: if* σ *is finite, then* $\mathrm{end}(\sigma) \vDash Q$*,*
- safety*: for all* $(i, t) \in \mathrm{dom}(\sigma)$*, if* $\sigma_{|i,t} \vDash A$*, then* $\sigma_{|i,t} \vDash G$*.*

Example 10 (one-way traffic, specification)**.** *The Hoare quintuple we want to specify to formalise the property in Example 7 is as follows:*

- $A \equiv (0 \le v_{\mathsf{POV}} \land -b_{\min} \le a_{\mathsf{POV}})$*,*
- $G \equiv (x < x_{\mathsf{POV}})$*,*
- $Q \equiv (v = 0)$*,*
- $P \equiv (x_{\mathsf{POV}} - x > \mathsf{dRSS}(v_{\mathsf{POV}}, v, \rho))$*.*

3 dFHL: a Hoare Logic for Automated Driving

In this section, we design dFHL (differential Floyd-Hoare logic) a logic to derive Hoare quintuples. We only present the derivation rule for dwhile, since other rules are very similar to standard ones for total correctness of Hoare triples.

To prove that some properties holds along the execution of a dwhile, we need to show that it is an invariant of the dynamics. In order to express that, we use the notion of *Lie derivative*.

The term $\mathcal{L}_{\dot{\mathbf{x}}=\mathbf{f}}\, e$ is called the *Lie derivative of* e *with respect to the dynamics* $\dot{\mathbf{x}} = \mathbf{f}$. If $\mathbf{x} = (x_1, \dots, x_n)$ and $\mathbf{f} = (f_1, \dots, f_n)$, its formal definition is

$$\mathcal{L}_{\dot{\mathbf{x}}=\mathbf{f}}\, e \ = \ \sum_{i=1}^{n} \frac{\partial e}{\partial x_i} f_i,$$

where the terms $\frac{\partial e}{\partial x_i}$ are the *partial derivatives of* e defined by induction e as usual.

The fundamental lemma of the Lie derivative (see [11]) is crucial for proving the soundness of the (DW) rule.

Lemma 11. *For any solution* $\hat{x} : \mathbb{R}_{\ge 0} \to \mathbb{R}^n$ *of the differential equations* $\dot{\mathbf{x}} = \mathbf{f}$*, the derivative of the function* $t \mapsto [\![e]\!]_{\hat{x}(t)}$ *is given by* $t \mapsto [\![\mathcal{L}_{\dot{\mathbf{x}}=\mathbf{f}}\, e]\!]_{\hat{x}(t)}$*.*

In other words, the value of expression e evolves according to $\mathcal{L}_{\dot{\mathbf{x}}=\mathbf{f}}\, e$ when variables evolve according to the dynamics $\dot{\mathbf{x}} = \mathbf{f}$.

We can now explain the rule in Figure 4. The idea is to find an *invariant*, a *variant*, and a *terminator*:

- the invariant is some property that stays unchanged along the dynamics, and which is used

$$\dfrac{\begin{array}{lll} \mathsf{inv}: & A \wedge e_{\mathsf{var}} \geq 0 \wedge e_{\mathsf{inv}} \sim 0 \Rightarrow \mathcal{L}_{\dot{\mathbf{x}}=\mathbf{f}}\, e_{\mathsf{inv}} \simeq 0 & \mathbf{Var}(e_{\mathsf{inv}}) \cap V_E = \emptyset \\ \mathsf{var}: & A \wedge e_{\mathsf{var}} \geq 0 \wedge e_{\mathsf{inv}} \sim 0 \Rightarrow \mathcal{L}_{\dot{\mathbf{x}}=\mathbf{f}}\, e_{\mathsf{var}} \leq e_{\mathsf{ter}} & \mathbf{Var}(e_{\mathsf{var}}) \cap V_E = \emptyset \\ \mathsf{ter}: & A \wedge e_{\mathsf{var}} \geq 0 \wedge e_{\mathsf{inv}} \sim 0 \Rightarrow \mathcal{L}_{\dot{\mathbf{x}}=\mathbf{f}}\, e_{\mathsf{ter}} \leq 0 & \mathbf{Var}(e_{\mathsf{ter}}) \cap V_E = \emptyset \end{array}}{A : [e_{\mathsf{inv}} \sim 0 \wedge e_{\mathsf{var}} \geq 0 \wedge e_{\mathsf{ter}} < 0]\ \mathsf{dwhile}\,(e_{\mathsf{var}} > 0)\,\{\dot{\mathbf{x}} = \mathbf{f}\}\ [e_{\mathsf{var}} = 0] : e_{\mathsf{inv}} \sim 0 \wedge e_{\mathsf{var}} \geq 0}\ (\mathrm{DW})$$

図 4 Hoare derivation rule for dwhile, here $(\sim, \simeq) \in \{(=,=), (>,\geq), (\geq,\geq)\}$

to prove the guarantee of the Hoare quintuple,

- the variant is some positive quantity that diminishes along the dynamics, and which is used to prove the postcondition,
- the terminator is used to prove termination of the dwhile.

To explain the rule a bit more in detail, we consider the case where $(\sim, \simeq) = (=,=)$. Then, we know from the precondition that $e_{\mathsf{inv}} = 0$ at the start of the execution, and by the top-left assumption, $\mathcal{L}_{\dot{\mathbf{x}}=\mathbf{f}}\, e_{\mathsf{inv}} = 0$, so the value of e_{inv} does not change along the dynamics, and therefore $e_{\mathsf{inv}} = 0$ always holds. We also know that $e_{\mathsf{var}} > 0$ throughout the dynamics (because of the operational semantics of the dwhile) and does not hold at the end, so $e_{\mathsf{var}} = 0$ by continuity at the end of the execution. This proves both the postcondition and the guarantee.

There are two subtle points. The first one is termination. In order to prove that the execution holds, we have to prove that the variant eventually reaches 0. To ensure that it decreases, it would be enough to show that $\mathcal{L}_{\dot{\mathbf{x}}=\mathbf{f}}\, e_{\mathsf{var}} < 0$, but since time and values are continuous, it is not enough to show that it reaches 0. That is the role of the terminator, which is some negative decrease quantity (it decreases by the bottom-left assumption). The left assumption thus shows that the variant decreases with at least a constant pace, so it eventually reaches 0.

The second subtlety is the behaviour of the environment. It could be that the changes to environment variables change the value of the invariant, the variant, or the terminator, which could break the rule. The right assumptions ensure that it is not the case.

Example 12 (one-way traffic, proof)**.** *We now want to prove the Hoare quintuple from Example 10. Since the structure of α is a composition of two* dwhile *commands, we need to apply the* (DW) *rule twice.*

For the first application, the invariant is $x - x_{\mathsf{POV}} - \mathsf{dRSS}(v, v_{\mathsf{POV}}, \rho - t) \geq 0$, the variant is $\rho - t$, and the terminator is -1.

For the second application, the invariant is $x - x_{\mathsf{POV}} - \mathsf{dRSS}(v, v_{\mathsf{POV}}, 0) \geq 0$, the variant is v, and the terminator is $-b_{\min}$.

4 dFHL$^{\downarrow}$: Reasoning about Safety Architectures

Formally proving properties of safety architectures is important to derive formal guarantees with our method. For example, since RSS can be enforced using the simplex architecture, we have to prove not only properties of RSS, but also how it interacts with other controllers in the simplex architecture to get some guarantees that can be used in practice.

4.1 Syntax

We first develop some syntactic constructs specifically tailored towards describing safety architectures and reasoning about them.

The base construct that we define is the *as-long-as* construct. The construct α *as long as* A, which we denote by $\alpha \downarrow A$, behaves as α as long as A holds, but as soon as A becomes false, the execution of α is interrupted. This construct can be defined as syntactic sugar by induction on α. From $\alpha \downarrow A$, we can derive more complex constructs that are useful to model safety architectures.

The first one is the *fallback* construct $\alpha \rightarrow_A \beta \equiv \big(\alpha \downarrow A; \mathsf{if}\,(\neg A)\,\{\beta\}\big)$, which behaves like α as long as A holds. If A becomes false at some point, then the execution of α is stopped and β is executed. In the case of automated driving, this constructs allows us to model a simple form of safety architecture, where α is a general controller, β a controller based on safety, and safety becomes critical when A becomes false. However, contrary to the simplex architecture, once β is executed, the general controller α cannot resume execution, even when safety is no longer critical.

The second construct is the *simplex* of α and β, which we denote by $\alpha \overset{B}{\underset{\rightarrow_A}{\leftarrow}} \beta$. Which behaves like $\alpha \rightarrow_A \beta$, but when B becomes false, the execution of α restarts. This models a simplex architecture, where B becoming false models the fact that the situation is no longer safety-critical. This construct can also be defined as syntactic sugar.

Example 13 (one-way traffic, simplex architecture)**.** *We want to study the scenario in Example 7, but where* SV *is controller by a safety architecture. We want to prove that, if the lead vehicle keeps its speed above some fixed* $v_{\min} > 0$*, then* SV *can reach a fixed target* x_{tgt} *while avoiding collisions. Otherwise, we can still avoid collisions.*

The programs of interest are α *and* β*, which are defined as follows:*

$$\delta_{\mathsf{brake}} \equiv \{\dot{x} = v, \dot{v} = -b_{\min}, \dot{x}_{\mathsf{POV}} = v_{\mathsf{POV}}, \dot{v}_{\mathsf{POV}} = a_{\mathsf{POV}}\},$$

$$\delta_{\mathsf{cruise}} \equiv \{\dot{x} = v, \dot{v} = 0, \dot{x}_{\mathsf{POV}} = v_{\mathsf{POV}}, \dot{v}_{\mathsf{POV}} = a_{\mathsf{POV}}\},$$

$$\alpha \equiv \begin{bmatrix} \mathsf{dwhile}\,(v > v_{\min} \wedge x < x_{\text{tgt}})\,\{\delta_{\mathsf{brake}}\}; \\ \mathsf{dwhile}\,(x < x_{\text{tgt}})\,\{\delta_{\mathsf{cruise}}\}; \\ \mathsf{dwhile}\,(v > 0)\,\{\delta_{\mathsf{brake}}\} \end{bmatrix},$$

$$\beta \equiv \mathsf{dwhile}\,(v > 0)\,\{\delta_{\mathsf{brake}}\}.$$

When POV *keeps its velocity above* $v_{\min}$ *at all times, then we execute* α*. Otherwise, we execute* β*. The whole behaviour of the system is thus modelled as the fallback* $\alpha \rightarrow_C \beta$*, where* $C \equiv (v_{\mathsf{POV}} > v_{\min})$*.*

4.2 Proof Rules

We want to prove two derivation rules for the fallback architecture. The idea is that, during an execution of $\alpha \rightarrow_C \beta$,

- either C holds throughout the entire execution, in which case only α runs, and we should be able to get a strong guarantee
- or C stops holding at some point, in which case β starts executing, and we can only get a weaker guarantee.

We call the first case the *strong assumption* case, because the strong condition C holds throughout the execution, and the second one the *weak assumption* case.

Similar rules can be proved for the simplex architecture as well as more complex architectures.

Strong Assumption

The strong assumption case is rather straightforward, since only α is ever executed, the Hoare quintuple about α will characterise the execution of the whole architecture.

We start by giving a rule about the as-long-as command.

Lemma 14 (strong as-long-as rule)**.** *This rule is correct:*

$$\frac{A : [P]\ \alpha\ [Q] : G \wedge C}{A : [P]\ \alpha \downarrow C\ [Q] : G.}\ (\downarrow_s)$$

Then we get the following rule for the fallback architecture.

Lemma 15 (Hoare rule for $\alpha \rightarrow_C \beta$ under strong assumption)**.** *If* $A : [P]\ \alpha\ [Q] : G \wedge C$ *is correct and* $A \wedge Q \Rightarrow G$ *then* $A : [P]\ \alpha \rightarrow_C \beta\ [Q] : G$ *is also correct.*

Weak Assumption

The weak assumption case is more subtle, since both α and β are executed. We first define *interruption-extensions*, which are properties that hold on an execution of $\alpha \to_C \beta$, even at the moment C becomes false.

Definition 16 (interruption-extension). *We say that assertion D is an* interruption-extension *(*int-ext *for short) of assertion C for program α from assertion P along assertion A if, for all $\rho \vDash P$, σ valid for α from ρ, and $(i,t) \in \mathrm{dom}(\sigma)$, if for all $(i',t') \in \mathrm{dom}(\sigma)$ such that $(i',t') < (i,t)$, $\sigma(i',t') \vDash A \wedge C$, and $\sigma(i,t) \vDash A$, then $\sigma(i,t) \vDash D$.*

This definition resembles the *safety* part of correctness in Definition 9 and states that, if C holds during the execution of α, except maybe at the end, then D holds at the end.

Interruption-extensions allow us to define a derivation rule for the as-long-as construct.

Lemma 17 (weak as-long-as rule). *This rule is correct:*

$$\frac{A \wedge C : [P]\ \alpha\ [Q] : G}{A : [P]\ \alpha \downarrow C\ [(Q \wedge C) \vee (D \wedge \neg C)] : D}\ (\downarrow)$$

where D is an int-ext of $G \wedge C$ for α from P along A.

The intuition is as follows. If C holds at all times, then α is not interrupted, and the assumption of this rule applies, so Q and G can be guaranteed. Otherwise α is interrupted, in which case, the assumption guarantees that C and G are true at all times except at the very last time. By definition of an int-ext, D is then guaranteed at all times.

Finally, we get a rule for the fallback architecture.

Lemma 18 (Hoare rule for $\alpha \to_C \beta$ under weak assumption). *If: $A : [P]\ \alpha\ [Q] : G$ and $A' : [P']\ \beta\ [Q'] : G'$ are correct, E is an int-ext of $G \wedge C$ for α from P along A', $E \Rightarrow P' \wedge G'$, $Q \Rightarrow Q'$, and $A' \wedge C \Rightarrow A$, and $Q' \Rightarrow G'$, then $A' : [P]\ \alpha \to_C \beta\ [Q'] : G'$ is correct.*

5 Conclusion

We have introduced our work on applying program logic dFHL to formally prove safety of automated driving scenarios. We have shown how to extend it to dFHL$^{\downarrow}$ to reason about safety architectures.

There are many things missing from this paper. Apart from the obvious missing rules, we have not explained the methodology we have developed to derive RSS conditions for driving scenarios (we have only shown how to prove that an assertion is actually an RSS condition), and how this scales to complex scenarios such as the emergency pull over scenario in Figure 3.

参 考 文 献

[1] Chan, C.-Y.: Advancements, prospects, and impacts of automated driving systems, *International journal of transportation science and technology*, Vol. 6, No. 3(2017), pp. 208–216.

[2] Crenshaw, T. L., Gunter, E., Robinson, C. L., Sha, L., and Kumar, P. R.: The Simplex Reference Model: Limiting Fault-Propagation Due to Unreliable Components in Cyber-Physical System Architectures, *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, 2007, pp. 400–412.

[3] Eberhart, C., Dubut, J., Haydon, J., and Hasuo, I.: Formal Verification of Safety Architectures for Automated Driving, *2023 IEEE Intelligent Vehicles Symposium (IV)*, IEEE, 2023, pp. 1–8.

[4] Hasuo, I., Eberhart, C., Haydon, J., Dubut, J., Bohrer, R., Kobayashi, T., Pruekprasert, S., Zhang, X.-Y., Pallas, E. A., Yamada, A., Suenaga, K., Ishikawa, F., Kamijo, K., Shinya, Y., and Suetomi, T.: Goal-Aware RSS for Complex Scenarios via Program Logic, *IEEE Transactions on Intelligent Vehicles*, Vol. 8, No. 4(2023), pp. 3040–3072.

[5] Liu, E. I., Pek, C., and Althoff, M.: Provably-Safe Cooperative Driving via Invariably Safe Sets, *2020 IEEE Intelligent Vehicles Symposium, IV 2020, Las Vegas, United States, October 19-22, 2020*, IEEE, 2020, pp. 8.

[6] Luo, Y., Zhang, X.-Y., Arcaini, P., Jin, Z., Zhao, H., Ishikawa, F., Wu, R., and Xie, T.: Targeting requirements violations of autonomous driving systems by dynamic evolutionary search, *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2021, pp. 279–291.

[7] Pek, C. and Althoff, M.: Efficient Computation of Invariably Safe States for Motion Planning of Self-Driving Vehicles, *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2018, pp. 3523–3530.

[8] Platzer, A.: *Logical Foundations of Cyber-Physical Systems*, Springer International Publishing, 2018.

[9] Seto, D., Krogh, B., Sha, L., and Chutinan, A.: The Simplex architecture for safe online control system upgrades, *Proceedings of the 1998 American Control Conference. ACC (IEEE Cat. No.98CH36207)*, Vol. 6, 1998, pp. 3504–3508 vol.6.

[10] Shalev-Shwartz, S., Shammah, S., and Shashua, A.: On a Formal Model of Safe and Scalable Self-driving Cars, *CoRR*, Vol. abs/1708.06374(2017).

[11] Trautman, A.: Remarks on the history of the notion of Lie differentiation, *Variations, Geometry and Physics: In honour of Demeter Krupka's sixty-fifth birthday*, Krupková, O. and Saunders, D. J.(eds.), Nova Science, 2008, pp. 297–302.