Formal Verification of an Approach for Systematic False Positive Mitigation in Safe Automated Driving Systems

Ayhan Mehmed
TTTech Auto AG
Vienna, Austria
ayhan.mehmed@tttech-auto.com

Wilfried Steiner
TTTech Computertechnik AG
Vienna, Austria
wilfried.steiner@tttech.com

Aida Čaušević
Mälardalen University
Västerås, Sweden
aida.causevic@mdh.se

***Abstract*—Manufacturers of self-driving cars need to significantly improve the safety of their products before the series of such cars are deployed in everyday use. A large number of architecture proposals for Automated Driving Systems (ADS) are aiming at addressing the challenge of safety. These solutions typically define redundancy schemes and quite commonly include self-checking pair structures, e.g., commander/monitor approaches. In such structures, the problem of detecting false positive failures arises, i.e., the monitor may falsely classify the output of the commander as being faulty. In this report we provide details regarding a formal verification of an approach aiming at false positive mitigation in the domain of automated driving. We formalize our proposal in an abstract model and prove the absence of false positives by means of k-induction.**

***Index Terms*—automated driving systems, run-time monitoring, false positive, congruency exchange**

I. Introduction

In our earlier work, we have introduced a novel method for reducing the false positive rate of run-time monitors implemented in a fail-safe (e.g., fail-silent, fail-operational) ADS architecture. In this report the proposed method is formalized in an abstract model and the absence of false positives is proved by means of k-inductions.

The report is organised as follows. In Section II we present background information presented, i.e., the assumed ADS architecture, the problem statement, as well the proposed solution. In Section III the proposed solution is formalized in an abstract model and the absence of false positives is proved by means of k-induction. Section IV provides final remarks and concludes the paper.

II. Background

A. A General ADS Fail-Silent Architecture

One way to handle the unsafe operation of ADS is by designing the system to be fail-silent, i.e., a design that shuts down system output upon detection of a failure. A common way to implement a fail-silent design is the *Commander-Monitor* architecture (also called Doer/Checker).

Figure 1 presents an example fail-silent ADS architecture known as *Commander-Monitor* architecture (or Doer/Checker architecture). The ADS consist of: (i) *Commander* (COM), that is the sub-system responsible for the AD functionality, (ii) *Monitor* (MON) for verifying the safe operation of COM. Based on the verification results, the MON decides whether to forward the COM output for execution to the actuators or not.

COM consists of *Pre-Processing* (PP-COM), *Sensor Fusion* (SF-COM), *Free Space Produces* (FSP-COM), and *Trajectory Planning* (TP-COM) builindg blocks. Furthermore, COM is additionally labeled with Fault-Containment Region 1 (*com_fcr1*), to indicate that any fault occurring inside the region will not propagate to other parts of the system. MON includes *Pre-Processing* (PP-MON), *Sensor Fusion* (SF-MON), and *Free Space Producer*-MON (FSP-MON) blocks, that perform similar functions as PP-COM, SF-COM and FSP-COM. However, as the MON is responsible for ensuring the overall ADS safety, it is expected that these components are less complex and safer than their counterparts in COM. Moreover, the MON includes the so-called CHECK-MON block that is responsible for (i) verifying the correctness of the COM output (i.e., *trajectory)* and (ii) based on the verification results to forward or discard them.

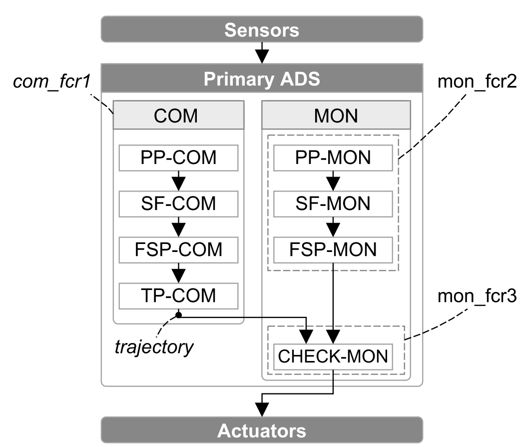

Fig. 1. An example of a fail-silent ADS architecture.

It is essential to note that the sub-components of MON are distributed into two fault-containment regions. *mon_fcr2* includes PP-MON, SF-MON, FSP-MON, whereas *mon_fcr3* includes CHECK-MON. The rationale for this is that CHECK-MON is assumed to be highly critical element that can be constructed as fail-silent fault-containment region, i.e., as ASIL D component according to ISO 26262.

Based on the objects detected by SF-MON, the FSP-MON calculates a space for safe trajectories called the *free_space_mon*. CHECK-MON then takes the output from TP-COM (i.e., the *trajectory*) and the FSP-MON (i.e., the *free_space_mon*) and verifies whether the *trajectory* is safe by checking whether it is located entirely inside the *free_space_mon*. In case of successful verification (i.e., the *trajectory* is verified to be in *free_space_mon*), CHECK-MON forwards the *trajectory* to the actuators. Otherwise, CHECK-MON shuts down the COM output by not passing the *trajectory* to the actuators, hence achieving a fail-silent behavior.

B. Problem statement

The fail-silent design presented in II-A has been proposed in [1] as part of a fail-operational ADS architecture. We assume a single failure hypothesis in which `com_fcr1`, `mon_fcr2`, and `mon_fcr3` fail independently from each other. The COM may fail arbitrarily. There are two types of MON failures, false negative and false positive detections. A false negative detection occurs when the MON falsely concludes that an unsafe trajectory is safe. A false positive occurs when the MON falsely concludes that a safety trajectory is unsafe. False negative detections are out of the scope of this paper and could be addressed by sufficient design diversity, ASIL decomposition, and certification processes, or a combination of these. In this paper, we focus on false positive detections, in particular, their mitigation.

Figure 2 describes the occurrence of an example of a false positive detection. In Figure 2 (a), the COM has generated a safe trajectory. In Figure 2 (b), the MON has generated the *free_space_mon* as described in Section II-A, however in this case leading to a false positive result occurring. The leading cause for the false positive, is the difference in the precision between COM and MON, as explained in Section II-A. Specifically, the approximation of OBJ1 made by MON (i.e., MON-RTO1) is less precise than the approximation made by COM (i.e., COM-RTO1). As a result, an otherwise safe trajectory is verified to be unsafe as it is not entirely inside the *free_space_mon*, and it goes into the non-drivable area surrounding OBJ1.

The frequent occurrence of such false positives is not desirable as it will significantly reduce the availability of the system and thus achieve opposite results compared to the initial goals of the better comfort in driving. Furthermore, in some cases, the false positive can as well affect the safety aspect of the ADS. Consider a system designed to brake whenever the MON identifies an unsafe trajectory (i.e., a design similar to Automated Emergency Braking (AEB) system). Initiating

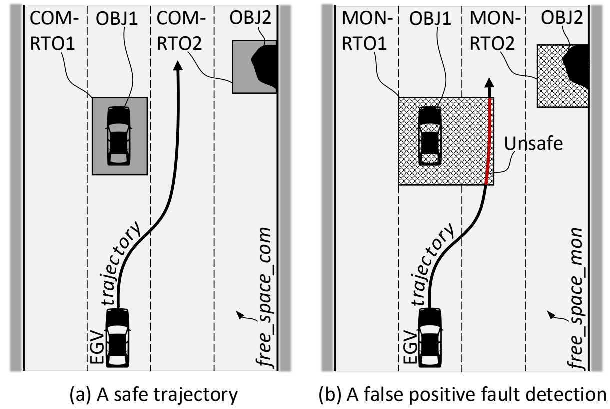

Fig. 2. An example of false positive.

sudden braking (due to a false positive) during an ordinary safe driving scenario may put the ego-vehicle into an unsafe situation. For example, the vehicle behind the ego-vehicle might not expect the sudden braking and crash to the ego-vehicle.

C. Proposed Solution

To address the false positive problem outlined in Section II-B, we introduce a novel fail-silent ADS architecture that reduces the false-positive rate of MON, while not reducing the overall ADS safety (see Figure 3).

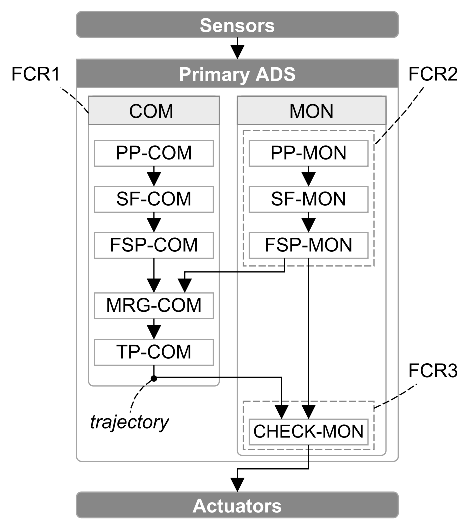

Fig. 3. A fail-silent ADS architecture with false positive reduction.

The architecture proposes two changes in comparison to the general fail-silent architecture in Section II-A. First, we introduce an additional block to the COM, namely the information merging block (MRG-COM). Second is the introduction of an information exchange channel between MON and COM, in particular from FSP-MON to the MRG-COM block [1]

[1] We assumes that the underlying architecture ensures real-time computation and communication that enables the coordinated exchange of information between MON and COM. Such architectures are [2], [3] [4], [5].

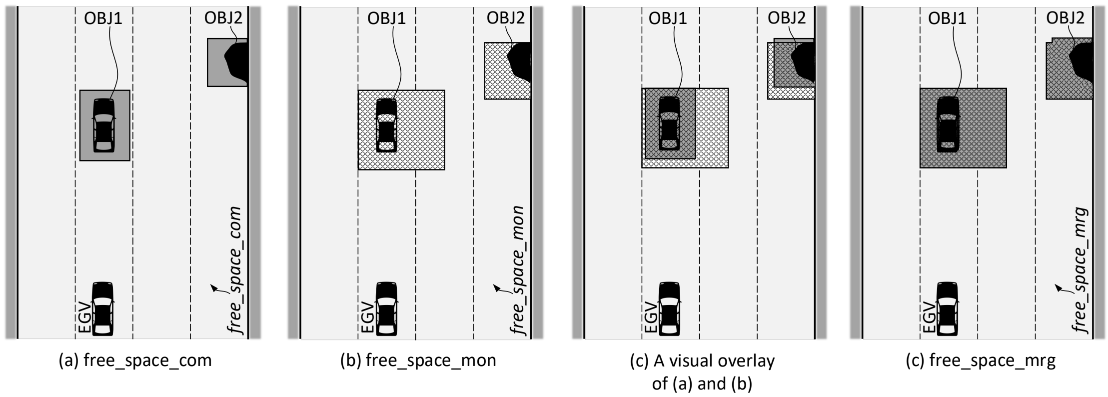

Fig. 4. Example merging process.

1) Merging Process: The MRG-COM block takes as an input the output of the FSP-COM and FSP-MON (i.e., *free_space_com* and *free_space_mon*) and provides a combination of these two outputs. The process is illustrated in Figure 4. Figure 4 (a) and (b) depict respectively the output from FSP-COM and FSP-MON. Figure 4 (c) is only for better reader experience and depicts the overlay of the free spaces generated from COM and MON. Figure 4 (d), presents the output of the MRG-COM block (i.e., *free_space_mrg*). To produce *free_space_mrg* the MRG-COM block combines *free_space_com* and *free_space_mon* using *set-theoretic cut-set* operation.

2) Trajectory planning based on merged free space: Following the architecture in Figure 3, the *free_space_mrg* is given to TP-COM that then accordingly plans a *trajectory* maneuvering the vehicle safely on the road (see Figure 5 (a)). As TP-COM needs to generate trajectories that fit *free_space_mrg* (which may be more conservative), the TP-COM trajectories may degrade in terms of comfort. The planned *trajectory* is going to the left-most lane, as all other three lanes in *free_space_mrg* are occupied.

3) Trajectory verification: Next, the generated *trajectory* from the TP-COM block is then sent to CHECK-MON block that then verifies the *trajectory* as described earlier in Section II-A. Figure 5 (b) illustrates the verification process. As can be seen, the *trajectory* is identified to be correct since it is completely in the *free_space_mon*. Hence, the CHECK-MON block will forward the *trajectory* to the actuators.

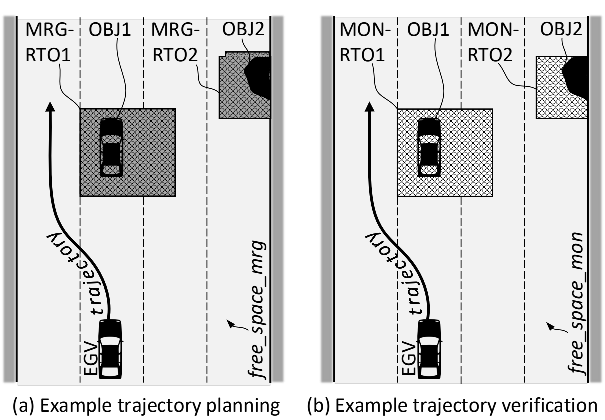

Fig. 5. An example trajectory planning based on *free_space_mrg* and trajectory verification.

A key element is the fact that the operation of the information merging stage MRG-COM is safe. That means even in case of failure of the MON providing a faulty *free_space_mon* to the commander COM, the information merging stage MRG-COM operation will not lead to a merged free space (i.e., *free_space_mrg*) that could cause a non-faulty TP-COM to produce an unsafe *trajectory*. We will argue the safety of the merging process more formally next.

III. Formal Verification

We explore the advantages of congruency exchange between COM and MON formally, by means of infinite bounded model-checking and k-induction.

Our proof and simulation framework is based on the bounded model checker for infinite-state systems that is part of the SAL environment. This model checker is called `sal-inf-bmc`. The algorithms presented in this paper have been formalized in SAL [6] as state-transition systems of the form $\langle S, I, \rightarrow \rangle$. S defines the set of system states σ_i, I is the set of initial system states with $I \subseteq S$, and $\rightarrow$ is the set of transitions between system states. Each system state σ maps state variables to particular values according to their defined type. The proof of an invariant property $\Box P$ ("P is always true") is done by k-induction [7], which is a generalized form of induction. k-induction consists of the following stages [8]:

- Base Case: Show that all the states reachable from I in no more than $k-1$ steps satisfy P
- Induction Step: For all trajectories $\sigma_0 \rightarrow \cdots \rightarrow \sigma_k$ of length k, show: $\sigma_0 \models P \wedge \cdots \wedge \sigma_{k-1} \models P \Rightarrow \sigma_k \models P$

We reuse a proof procedure proposed by Rushby [9]. An overview of the proof structure is given in Figure 6, where we model an ideal AD system as well as a real AD system and show that their input-output behavior equals under certain constraints. In addition to the ideal and the real AD system, we make our assumptions explicit by encoding them in an assumption module as well. In the following, we will incrementally construct the depicted proof.

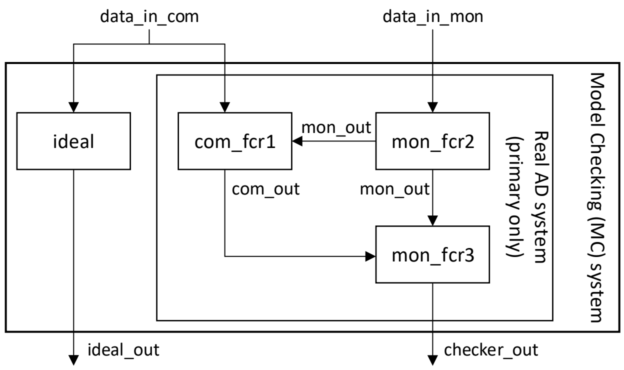

Fig. 6. Overview of the final proof structure

The construction of the model and the formal proof took an expert in the SAL language less than a month. The actual verification time of the properties is in the order of seconds on a standard laptop.

A. A Basic Formal Model

We use the SAL infinite bounded model checker and start the formal model with a set of TYPE definitions and some initial constants.

```
sensor_data: TYPE;
free_space: TYPE; free_space_init: free_space;
trajectory: TYPE; trajectory_init: trajectory;
trajectory_empty: trajectory;
```

`sensor_data` is an abstract type that represents sensor data - it is irrelevant which sensors are used. `free_space` is a type that represents free space, i.e., the space that the vehicle is able to maneuver with a sufficiently high probability of absence of collisions. We also define an initial free space as `free_space_init`. Finally, we define a type `trajectory` that represents trajectories and define two instances `trajectory_init`, an initial trajectory, as well as `trajectory_empty` as a constant that identifies when the AD system is not providing a trajectory.

We, furthermore, define some generic functions.

```
pp_sf_fsp_com(x: sensor_data): free_space;
pp_sf_fsp_mon(x: sensor_data): free_space;
tp_com(x: free_space): trajectory;
```

The function `pp_sf_fsp_com` combines the preprocessing, sensor fusion, and free space generation procedures of the COM. It takes as an input the sensor data and returns a free space for further trajectory planning, which is done by the function `tp_com`. The latter function takes as an input the free space and returns a trajectory. The combination of preprocessing, sensor fusion, and free space generation procedures in the MON is represented by the function `pp_sf_fsp_mon`. We assume that the COM actually produces a free space interpretation. However, the proofs can easily be transferred in case the COM would directly produce trajectories from sensor data without explicit representation of its perceived free space.

With these definitions, we are already able to express the ideal behavior of our AD system, the `ideal` AD system.

```
ideal: MODULE =
BEGIN
INPUT data_in_com: sensor_data
OUTPUT ideal_out: trajectory
INITIALIZATION ideal_out = trajectory_init;
TRANSITION
  ideal_out' =
    tp_com(pp_sf_fsp_com(data_in_com));
END;
```

The `ideal` AD system never fails and always returns a safe trajectory, i.e., a trajectory that does not cause a severe incident. We model this ideal behavior by a SAL module that takes as input some sensor data `data_in_com`, and returns as an output said safe trajectory `ideal_out`. To do so, we model the initial output to be a safe `trajectory_init` and further model the update process of the trajectories by a simple transition using the functions previously defined.

The `ideal` AD system behaves perfectly, yet we cannot build such a system in reality since we must consider the presence of failures in artificial components. Thus, we now model a real AD system that consists of components that, indeed, may fail.

This real AD system consist of a COM that is implemented by one fault-containment region `com_fcr1` and a MON that is implemented by two fault-containment regions `mon_fcr2` and `mon_fcr3` (see Figure 3 for the contents of fault-containment regions). We will further show by means of model-checking and k-induction that the behavior of this real AD system equals the behavior of the `ideal` AD system under certain conditions. It is the aim of this formal study to make these conditions explicit and to study the behavior of the real AD system in those cases when the previously described conditions do not hold (and therefore the behavior of the real AD system deviates from the `ideal` AD system).

We start with the first model of the commander `com_fcr1` (which we will refine later as the paper progresses).

```
com_fcr1: MODULE =
BEGIN
INPUT data_in_com: sensor_data
OUTPUT com_out: trajectory, com_error: BOOLEAN
INITIALIZATION com_out=trajectory_init; com_error=FALSE
TRANSITION
[ TRUE -->
  com_out' = tp_com(pp_sf_fsp_com(data_in_com));
  com_error' = FALSE;
[] TRUE -->
  com_out' IN {x: trajectory |
          x/=tp_com(pp_sf_fsp_com(data_in_com))};
  com_error' = TRUE;
] END;
```

This first `com_fcr1` module extends `ideal` in the following way: we introduce a second output `com_error` that is a boolean variable with initialization set to `FALSE`, i.e., we assume that initially the commander is non-faulty. We also extend the one transition from `ideal` with a second transition in `com_fcr1`. Note that SAL formulates transitions as guarded commands of the form `guard --> command`. As depicted, both `com` transitions have the guard set to true. Thus, in each execution step `com` is free to non-deterministically chose either one of the transitions. The first transition represents the failure-free execution of com_fcr1 and therefore equals the transition of ideal. The second transition represents a faulty execution of com_fcr1 that leads to com_fcr1 returning a trajectory that is different from the correct trajectory (however this faulty trajectory might be a safe trajectory as well). When com_fcr1 executes the second transition the model marks the occurence of a failure of com_fcr1 by setting com_error to TRUE. Note, we use com_error only to reason about the model, not as a part of an algorithm that the real AD system executes - we do not assume a faulty component to report its failure in operation.

We aim to verify that ideal and com_fcr1 return the same output, by the following system composition (MC_ideal simply executes ideal and com_fcr1 in parallel) and a lemma.

```
MC_ideal: MODULE = ideal || com_fcr1;
CTR_ideal: LEMMA MC_ideal |- G(ideal_out = com_out);
```

It should not come as a surprise that the verification attempt of the CTR_ideal lemma fails, since we explicitely allowed com_fcr1 to produce trajectories that differ from ideal (that is when com_fcr1 fails). However, if we assume that com_fcr1 does not fail, then the verification attempt should be successful. We can formally express this assumption by an additional module assumptions_simple.

```
assumptions_simple: MODULE =
BEGIN
INPUT com_error: BOOLEAN
OUTPUT a_hold: BOOLEAN
DEFINITION a_hold = IF com_error THEN FALSE ELSE TRUE ENDIF;
END;
```

The assumptions module takes com_error as an input and returns a boolean a_hold as output that indicates whether the assumptions hold or not. In this simple case we are interested in the assumption that com_fcr1 is not faulty. We can then verify that com_fcr1 and ideal, indeed, have identical output as follows.

```
MC_a_simple: MODULE=ideal||com_fcr1||assumptions_simple;
PROOF_a_simple: LEMMA MC_a_simple |-
      G(a_hold => ideal_out = com_out);
```

We include the assumptions_simple model in the system composition and extend the lemma by the hypothesis that a_hold is true. Then, indeed, k-induction verifies this lemma. While this is not a major result, we have introduced the full verification framework used in the following study in which we modify our assumptions in the assumptions module, as we update the real AD system, with the monitor fault-containment regions mon_fcr2 and mon_fcr3.

B. Model Updates of the real AD system

We begin with the fault-containment region *mon_fcr2*.

```
mon_fcr2: MODULE =
BEGIN
INPUT data_in_mon: sensor_data
OUTPUT mon_out: free_space, mon_error: BOOLEAN
INITIALIZATION mon_out = free_space_init; mon_error=FALSE;
TRANSITION
[ TRUE -->
  mon_out' = pp_sf_fsp_mon(data_in_mon);
  mon_error' = FALSE;
[] TRUE -->
   mon_out' IN
     {y: free_space | y/=pp_sf_fsp_mon(data_in_mon)};
   mon_error' = TRUE;
] END;
```

The fault-containment region mon_fcr2 of the monitor shares many similarities with com_fcr1. The main difference being that mon_fcr2 does not generate a trajectory, but only returns a free space via its output mon_out. This free space is the result of the mon_fcr2 function pp_sf_fsp_mon that uses sensor data for the mon_fcr2 as the input. In analogy to com_fcr1 also mon_fcr2 defines two transitions. A first transition for the failure-free case and the second transition for the failure case. In the latter case, mon_fcr2 returns a faulty free-space, i.e., a free space that is different from the free space it would have generated in the failure-free case. Similar to the com_fcr1, the faulty free space is not necessarily unsafe (we will continue this discussion a little later in this paper).

The checker CHECK-MON is modeled by the mon_fcr3 module and uses the following trajectory_verified? predicate.

```
trajectory_verification(x: free_space): trajectory;
trajectory_verified?(x: trajectory, y: free_space):
  BOOLEAN = IF x = trajectory_verification(y)
            THEN TRUE ELSE FALSE ENDIF;
```

trajectory_verified? returns TRUE, if a given trajectory *x* matches a given free_space *y*, and FALSE otherwise. We will later use the trajectory_verified? predicate in mon_fcr3 and in assumptions or the com_fcr1 module. For the architectural properties we are interested to analyze, it is not important to be specific on the semantics of this evaluation, but it is rather important that we assume the existence of such an evaluation. mon_fcr3 then simply uses this predicate.

```
mon_fcr3: MODULE =
BEGIN
INPUT com_out: trajectory, mon_out: free_space
OUTPUT checker_out: trajectory
INITIALIZATION checker_out = com_out;
TRANSITION
  checker_out' = IF trajectory_verified?(com_out', mon_out')
            THEN com_out' ELSE trajectory_empty ENDIF;
END;
```

mon_fcr3 takes the output of com_fcr1 and mon_fcr2 (i.e., com_out and mon_out) as an input. It uses the predicate trajectory_verified? to check the compatibility of the trajectory and the free space and returns either the com_out trajectory (in case of successful evaluation) or an empty trajectory trajectory_empty (in case if the trajectory and the free space do not match). In case the checker returns trajectory_empty, the real AD system switches to a backup system.

We also need to update the assumptions to reflect a *single-failure hypothesis* as well as our hypothesis that a correct com_fcr1 only produces trajetories that are verified by a correct mon_fcr2. As discussed earlier, there are multiple ways to tolerate the failure of the checker mon_fcr3. For simplicity reasons, in this paper (and proofs) we assume that mon_fcr3 fails silent, i.e., in the failure case mon_fcr3 returns the empty trajectory trajectory_empty.

```
assumptions_2: MODULE =
BEGIN
INPUT com_error: BOOLEAN, mon_error: BOOLEAN,
  com_out: trajectory, mon_out: free_space
OUTPUT a_hold: BOOLEAN
DEFINITION
  a_hold = IF ((NOT com_error AND NOT mon_error) OR
              (    com_error AND NOT mon_error) OR
              (NOT com_error AND     mon_error)) AND
              ((NOT com_error AND NOT mon_error)
                => trajectory_verified?(com_out, mon_out))
   THEN TRUE ELSE FALSE ENDIF;
END;
```

We can then compose the system again and verify a more interesting property as follows.

```
ADS: MODULE = com_fcr1 || mon_fcr2 || mon_fcr3;
MC_a2: MODULE = ideal || ADS || assumptions_2;
PROOF_a2: LEMMA MC_a2 |-
G(a_hold => (ideal_out=checker_out OR
  checker_out=trajectory_empty OR
   com_error AND NOT mon_error AND
   trajectory_verified?(com_out, mon_out)));
```

Indeed, we can verify the PROOF_a2 lemma presented above, and show the benefit of Rushby's formal approach quite well. We want to prove that given that our assumptions hold, there is a certain relation between the ideal AD system (the one that does not fail) and the real AD system (that does fail). Indeed, by this formal notation and model-checking, we discover that we need to distinguish three cases:

- ideal_out=checker_out: this is the failure-free case in which the real AD system behaves as the ideal AD system.
- ideal_out=trajectory_empty: this is the behavior in the failure case that we would expect. When either com_fcr1 produces a faulty trajectory or mon_fcr2 produces a faulty free-space that fails to verify the commander's trajectory then the checker returns an empty trajectory.
- com_error AND NOT mon_error AND trajectory_verified?(...): in this case, com_fcr1 is faulty and produces a trajectory that diverges from ideal. However, this faulty trajectory is still accepted by mon_fcr3. This third case, although not so obvious, has a natural interpretation: com_fcr1 may be faulty, but could actually, still produce a trajectory that is good enough to be accepted by mon_fcr3.

Note that the formal model-checking process has been guiding us towards this third case. When we aim to verify the PROOF_a2 property without this third case, the model checker returns a counterexample from which we can deduct this third case.

C. Assumptions Reduction

The assumptions_2 are actually quite demanding, since they state perfection of a non-faulty mon_fcr3 to classify all trajectories from a non-faulty com_fcr1 as correct. Without explicit coordination mechanism between com_fcr1, and mon_fcr2 this will be hardly achievable: diversity in sensor inputs, technology choices, and/or algorithmic realizations, as well as, clock skews, and other synchronization inaccuracies will lead to false positives with quite reasonable probability. It is likely that the checker mon_fcr3 falsely identifies good trajectories by the commander as being faulty.

Alternatively to these stringent assumptions we can explicitly design a coordination activity between com_fcr1 and mon_fcr2. We can forward the free space calculated by mon_fcr2 to com_fcr1 and com_fcr1 may be designed in a way such that it generates trajectories that are in the com_fcr1 free space **as well as** in the mon_fcr2 free space. From a modeling perspective this can be easily achieved by updating the com_fcr1 module to the com_fcr1_w_mrg module.

```
com_fcr1_w_mrg: MODULE =
BEGIN
INPUT data_in_com: sensor_data,
  mon_out: free_space, mon_error: BOOLEAN
OUTPUT
  com_out: trajectory, com_error: BOOLEAN
INITIALIZATION
  com_out = trajectory_init; com_error = FALSE;
TRANSITION
[ TRUE -->
  com_out' IN {x: trajectory |
    x=tp_com(pp_sf_fsp_com(data_in_com)) AND
      trajectory_verified?(x, mon_out')
       OR (mon_error' AND x=trajectory_empty)
       OR (mon_error' AND x/=tp_com(pp_sf_fsp_com(data_in_com)};
  com_error' = FALSE;
[] TRUE
  -->
  com_out' IN {x: trajectory |
x /= tp_com(pp_sf_fsp_com(data_in_com))};
  com_error' = TRUE;
] END;
```

We provide com_fcr1_w_mrg as an additional input to the free space from mon_fcr2 and require in the correct operation of com_fcr1_w_mrg that the trajectory calculated satisfies also the mon_frc2 free space. Practically, this can be achieved by calculating the cut set of com_fcr1_w_mrg and the mon_frc2 free spaces and having com_fcr1_w_mrg use this cut-set for trajectory planning.

There is, however also the possiblity that com_fcr1_w_mrg is not able to calculate such a trajectory (i.e., a trajectory that satisfies both free spaces). Yet, this can only be the case when com_fcr1_w_mrg or mon_fcr2 is faulty, i.e., in case of correctly operating com_fcr1_w_mrg and mon_fcr2 such a cut set can be calculated.

In order to reflect this impossibiltiy to generate a trajectory to the mon_fcr2 execution in the failure case mon_fcr2 needs to inform com_fcr1_w_mrg of mon_error. Of course, in a real implementation the input mon_error would not be available to the commander, and merely the fact that no trajectory is produced would be observed. Since we do not explicitely model a trajectory generation function we need to add the mon_error input to com as an auxiliary modeling artefact to avoid an overly optimistic model, i.e., the optimistic model would always return a trajectory, even if no trajectory exists that satisfy the cut set of com_fcr1_w_mrg and a mon_fcr2 free space.

It could also be the case that a faulty monitor mon_fcr2 would prevent the commander com_fcr1 to produce the same trajectory as the ideal AD. However, the commander com_fcr1 could still find a trajectory that is safe. This case is indicated by x/=tp_com(pp_sf_fsp_com(data_in_com) above.

With such an additional coordination mechanism, we can relax the assumptions to assumptions_3.

```
assumptions_3: MODULE =
BEGIN
...
DEFINITION
  a_hold = IF ((NOT com_error AND NOT mon_error) OR
               (    com_error AND NOT mon_error) OR
               (NOT com_error AND     mon_error))
           THEN TRUE ELSE FALSE ENDIF;
END;
```

We can then verify the following lemma, accordingly.

```
MC_a3_mrg: MODULE = ideal || ADS_w_mrg || assumptions_3;
PROOF_a3_mrg_1: LEMMA MC_a3_mrg |-
G(a_hold => (ideal_out=checker_out OR
  checker_out=trajectory_empty OR
  (com_error AND NOT mon_error AND com_out/=ideal_out
    AND trajectory_verified?(com_out, mon_out))));
```

The PROOF_a3_mrg_1 is essentially equal to the PROOF_a property that we proved before, however in a different system model. We translated the strong assumptions necessary to verify the PROOF_a property to an additional functionality in the com_fcr1_w_mrg module.

Finally, we can also prove that false positives are not possible.

```
PROOF_a3_mrg_3: LEMMA MC_a3_mrg |-
  G(NOT com_error =>
    (checker_out=ideal_out OR
(mon_error AND checker_out=trajectory_empty) OR
(mon_error AND com_out/=ideal_out AND
   trajectory_verified?(com_out, mon_out))));
```

The PROOF_a3_mrg_3 property verifies that as long as the com_fcr1_w_mrg operates correctly, the real AD system behaves as the ideal AD system. However, in the case of a mon_fcr2 failure, the real AD system may fail to produce a trajectory (i.e., mon_fcr3 failures are assumed to produce empty trajectories as well) or produces another safe trajectory.

We summarize the considered failure scenarios in Table I. Each row represent a particular type of settings. In each setting (i.e., row) the table indicates whether an FCR is fault-free or faulty. In case a FCR is faulty we name the failure mode. Each setting also indicates the overall output of the ADS. The last column indicates whether the setting has been verified in the formal model or by informal argument.

com_fcr1	mon_fcr2	mon_fcr3	ADS output	Coverage
fault-free	fault-free	fault-free	ideal_out	Formal Model
unsafe trj	fault-free	fault-free	trj_empty	Formal Model
faulty-safe	fault-free	fault-free	trj!=ideal	Formal Model
fault-free	non-drivable	fault-free	trj_empty	Formal Model
fault-free	faulty-drivable	fault-free	traj!=ideal	Formal Model
fault-free	fault-free	fail-silent	trj_empty	Informal

TABLE I
ADDRESSED FAILURE SCENARIOS

IV. CONCLUSIONS

ADS developed for L3 and above automated driving are highly safety-critical and thus must ensure safe operating during their entire lifetime. The design of ADS with sufficient levels of safety and availability is a challenge. One design measure towards such acceptable ADS is to design appropriate run-time monitoring mechanisms, such to construct commander/monitor pairs (also known as doer/checker). Thus, run-time monitoring, applied in a commander/monitor ADS architecture, provides an on-going verification of the safe operation of the commander. Important performance characteristics of run-time monitors are their false positive and false negative rates.

In this report we have (i) described a solution for removing false positives in a systematic way (ii) and have formalized the solution in an abstract model to prove the absence of false positives by means of k-induction.

REFERENCES

[1] A. Mehmed, W. Steiner, M. Antlanger, and S. Punnekkat, “System architecture and application-specific verification method for fault-tolerant automated driving systems,” in *2019 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2019, pp. 39–44.

[2] P. Caspi, C. Mazuet, and N. R. Paligot, “About the design of distributed control systems: The quasi-synchronous approach,” in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2001, pp. 215–226.

[3] R. Larrieu and N. Shankar, “A framework for high-assurance quasi-synchronous systems,” in *2014 Twelfth ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE)*. IEEE, 2014, pp. 72–83.

[4] H. Kopetz and G. Bauer, “The time-triggered architecture,” *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, 2003.

[5] A. Benveniste, P. Caspi, P. Le Guernic, H. Marchand, J.-P. Talpin, and S. Tripakis, “A protocol for loosely time-triggered architectures,” in *International Workshop on Embedded Software*. Springer, 2002, pp. 252–265.

[6] L. de Moura, S. Owre, H. Ruess, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari, “SAL 2,” in *Computer-Aided Verification)*, ser. LNCS, vol. 3114. Springer, 2004, pp. 496–500.

[7] L. de Moura, H. Rueß, and M. Sorea, “Bounded model checking and induction: From refutation to verification,” in *Computer-Aided Verification, CAV 2003*, ser. Lecture Notes in Computer Science, vol. 2725. Springer, 2003, pp. 14–26.

[8] B. Dutertre and M. Sorea, “Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata,” in *Proc. of FORMATS/FTRTFT*, ser. Lecture Notes in Computer Science, vol. 3253. Springer-Verlag, Sep. 2004, pp. 199–214.

[9] J. Rushby, “A safety-case approach for certifying adaptive systems,” in *AIAA Infotech@ Aerospace Conference and AIAA Unmanned... Unlimited Conference*, 2009, p. 1992.