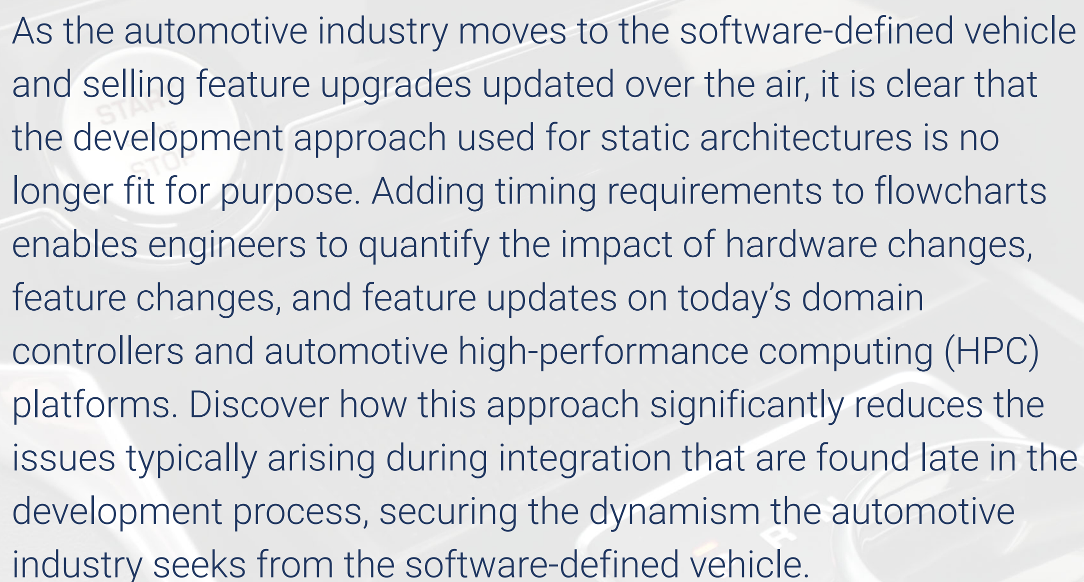
WHITEPAPER

INCHRON
THINK REAL-TIME

Why an Event-Chain-Centered Architecture Design Is the Right Response to the Challenges of Developing Software-Defined Vehicles

Author: Dipl. Inf. Florian Mayer, INCHRON AG



As the automotive industry moves to the software-defined vehicle and selling feature upgrades updated over the air, it is clear that the development approach used for static architectures is no longer fit for purpose. Adding timing requirements to flowcharts enables engineers to quantify the impact of hardware changes, feature changes, and feature updates on today’s domain controllers and automotive high-performance computing (HPC) platforms. Discover how this approach significantly reduces the issues typically arising during integration that are found late in the development process, securing the dynamism the automotive industry seeks from the software-defined vehicle.

Why an event-chain-centered architecture design is the correct response to the challenges of developing software-defined vehicles

The automotive industry is transitioning from today's predominantly domain-oriented E/E architecture toward a zonal architecture. This provides OEMs with an approach that enables greater flexibility that, thanks to a focus on services and high-performance compute platforms (HPC), ensures continuous product maintenance over a vehicle's lifetime. Short development cycles detached from the traditional vehicle development approach, coupled with over-the-air updates, will deliver on this vision of the software-defined vehicle, enabling functionality to be added by users in the form of upgrades after purchase.

Function nets cannot remain the supreme structure by which E/E architectures are organized

With software already accounting for the vast majority of added value in the vehicle, hardware dependency must be pushed as far back in the development process as possible. In fact, it must be possible to define and develop software almost independently of the hardware. This also affects requirements management, something that is currently still far too oriented towards static architectures, often in the form of function nets. Although this remains indispensable for decomposing the overall system into its logical and technical components of software and hardware, it is already the result of a much more decisive step: The analysis of the processes in the system that are necessary to achieve a specific customer function. Finally, many such functions must be allocated to the available resources in the finished vehicle without causing conflicts.

The actual know-how of vehicle manufacturers lies in their knowledge, built up through years of experience and systematic collection of user feedback, as to which dynamic requirements can be experienced and differentiated by their customers. For example, how many milliseconds should elapse between pressing a control and the acoustic, haptic, and visual feedback? After what time should a lane-changing automated driving function initiate a lane change? And how should this time vary between the sportier and more comfort-focused driver?

Even these limited examples highlight how a single system component, typically implemented as a software processing step, is reused in several scenarios or customer functions to satisfy completely different requirements. Additionally, that processing step can range from simple, such as controlling a turn signal, to highly complex, such as the selection of an optimal driving maneuver.

With requirements management approaches oriented only to the structure of the function net, it is impossible to capture these varying context-sensitive requirements separately and independently of one another. As a result, continuous maintenance and traceability through to testing are virtually impossible [1] .

Function nets mix up the causal relationships of most logical processing steps, thereby obscuring the view of the causal relationships between the function blocks. The signal networks between the blocks may even contain cycles that would be missing in a purely causal view (Figure 1). It is not possible to unambiguously determine an end-to-end latency without decomposing the network back into its causal relationships. So why not start with the causal relationships in the requirements management and link them to the functional network via trace links?

Figure 1:
Example of a function net (black) with two causal event chains, EC1 (blue) and EC2 (red).
While the function net contains a cycle, neither EC1 nor EC2 is cyclical.

How event-chain-centered architecture design solves the problem

Event chains describe causal relationships between the steps previously described by decomposing a behavior. In software, but also in system design, we often speak here of 'processing steps in a sequence.'

Let us consider the system 'hazard warning lights' and how event chain analysis affects the precision of the dynamic requirements. The first obvious event chain handles the switching-on of the hazard light feature. The second event chain deals with the cyclical switching of all the lamps. Both aspects can be experienced by the vehicle's user and are therefore of central importance for the system.

Event chains enable the precise analysis of the causal relationships involved in each scenario and thus also the assignment of requirements to the event chain step in the context of the scenario. For example, during the switch-on of the indicator LEDs, the software component's latency is important. However, when it comes to controlling the flashing, the synchronicity between the lamps during ongoing operation is of primary concern.

Event chains thus enable the separation of static and dynamic architecture and are key to the following critical architecture design patterns:

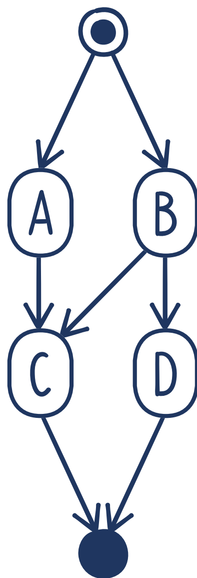
- **Separation of Concerns:** All important requirements for dynamic behavior can be clearly assigned at the event chain steps in requirements management, and can even be maintained there at a logical level independent of static decomposition as provided by a function net. This also brings the valuable know-how of the vehicle manufacturers one step further removed from the technical level.
- **Modularity:** Essential for software-defined vehicles, but already being driven by today's platform-thinking with variants, it is necessary to enable or disable various customer functions within a project. In addition, there is the aspect of multiple use: Static components (e.g., software components that implement specific processing steps) can be required several times in different customer functions, or even variants thereof, but must still meet specific requirements in each use case (reused software components).
- **Test and Verification:** Since requirements are assigned to event chain steps, this means they can be assigned for testing to precisely the test case, location, and time for which they are also relevant or verifiable. This leads to a consistent derivation of test cases, enabling a path from the requirement to its acceptance to be established.

Why sequence diagrams quickly reach their limits

Unfortunately, SysML sequence diagrams are only suited to simple descriptions of many dynamic processes. Certainly, they support modeling control flows such as alternative paths, loops, or parallel execution. So, there is no question of whether sequence diagrams can describe combinations of different flows. However, as seen in Figure 2, they rapidly reach their limits.

The dependency graph shows that the final node (bottom) can only be reached when all its predecessors, here C and D, have been reached. But these, in turn, depend on A and B. The challenges associated with describing all possible sequential processes in a sequence diagram quickly materializes as the results become unnecessarily complex and challenging to read.

DATA DEPENDENCY GRAPH



SEQUENCE DIAGRAM

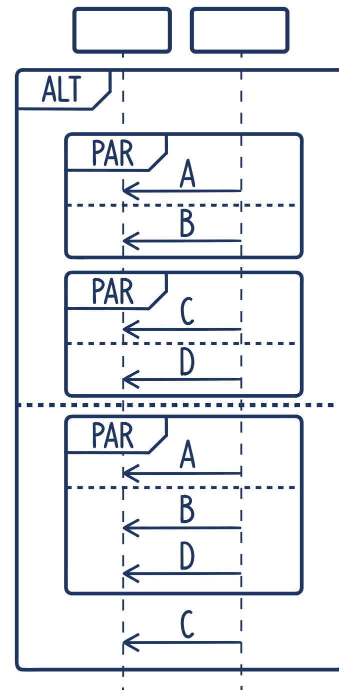


Figure 2:
Example of dependency graphs (left) and the corresponding sequence diagram (right) that, as a result, has become complex to read.

Flowcharts are an effective tool for effective event-chain-centric architecture design

Flowcharts, as defined by the OMG[2], are well suited to record the previously mentioned aspects of an event chain in a formal and machine-readable way, even for complex scenarios. They are well known, supported by a wide range of architectural tools, easy to understand, and, like sequence diagrams, offer formally specified execution semantics that makes them easy to simulate.

When compared to the classical, purely linear form of event chains, they have the advantage that they provide the linguistic means needed to describe almost any arbitrarily complex process. As we will show later, they are particularly helpful to phrase timing requirements to assure correct behavior on complex patterns like the safety supervisor.

Analogous to the separation of dynamic and static architecture, flowcharts enable the separation of the description of causal relationships from the process or function calls executed at the technical level that is subject to scheduling.

For example, if a function is permanently assigned to a specific process within a specific period, it is executed cyclically. However, due to the execution semantics of the associated event chain, an event chain step that refers to this function is only switched on when the causally necessary previous steps have also been observable in the system since that event chain's start. In this way, it is possible to start a timer when the trigger event for the event chain occurs to then measure the respective time in the context of the event chain for the function calls that causally follow. In this way, the time required for an emergency braking function to reach the desired braking pressure from the sudden occurrence of an external obstacle can be specified precisely in a machine-readable form.

Methods for simplifying the design of complex functions

A vital tool in the development team's toolbox is the ability to separate function from implementation. As far as possible, the implementation should also be independent of the hardware architecture. This is known as separation of concerns.

Consider the customer-function "advanced driving system" (ADS). This customer function employs several sub-components that are divided further into sub-functions. ADS_Planner, one of these sub-functions, cyclically delivers a new path to the vehicle's actuators. The safety supervisor (SSV), a sub-component, adds redundancy and diverse software design for accident prevention for safety reasons (Figure 3). This is achieved by comparing the path provided by the ADS_Planner with its own analysis of the traffic situation, based on less complex sensor information, with the only aim to foresee an imminent collision when continuing to follow that path.

If it detects such a threat, it would switch to an alternative safe path. More on the safety supervi-sor pattern can be found in work by M. Törngren[3] and other similar sources.

This is analogous to a learner driver failing their driving test when the driving instructor has to intervene. Such a path switch is considered an error in the system and would lead to a safety shutdown, which is, of course, to be avoided. Therefore, it is crucial to coordinate the timing between the ADS Planner and the SSV so that the ADS Planner can handle dangerous situations independently and without unrequired interventi-on by the SSV.

Without an event-chain-driven architecture design, this circumstance only becomes appa-rent at the technical level because this is when the actual runtimes and cycle times of the tasks become known. Integrators cannot tackle this challenge because the causal dependencies between the functional chain steps are unknown at their stage in the development process. Thus, they are unable to identify and eliminate such issues systematically.

By introducing event chains, complex timing requirements can be added to capture and avoid these situations. As can be seen in Figure 3, timing requirements can be formulated based on the steps of an activity chart. This timing require-ment assures that, after a concrete sensing response from Sensor A has been acquired, Sfty_Fusion should not complete before ADS_-Planner, thus assuring that no safety shutdown due to a timing issue can occur.

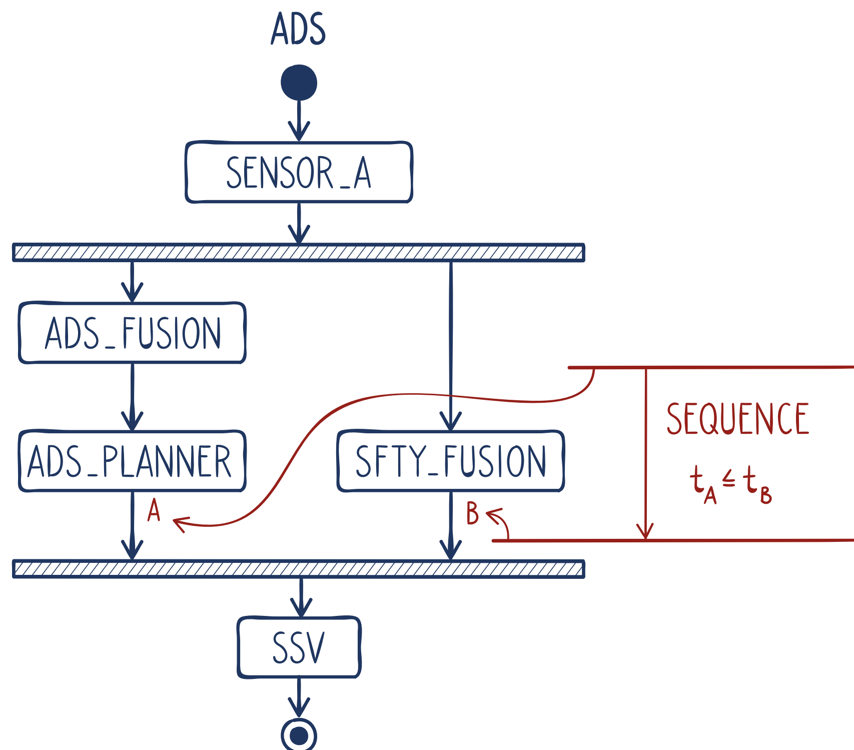


Figure 3:
Example of a synchronicity timing requirement:
“The ADS_Planner must complete before Sfty_Fusion”

Figure 4 shows a timing sequence that, from the integrator's point of view, looks acceptable because all tasks can be processed within their period. However, the problem of inconsistent data between ADS and SSV materializes at time instance $t_{sfty_shutdown}$. SSV starts, but ADS_Planner has not yet processed the most recent sample from Sensor_A. Therefore, the SSV will initiate a safety shutdown although the ADS_Planner completes its computation in time (with regards to its deadline determined by its period).

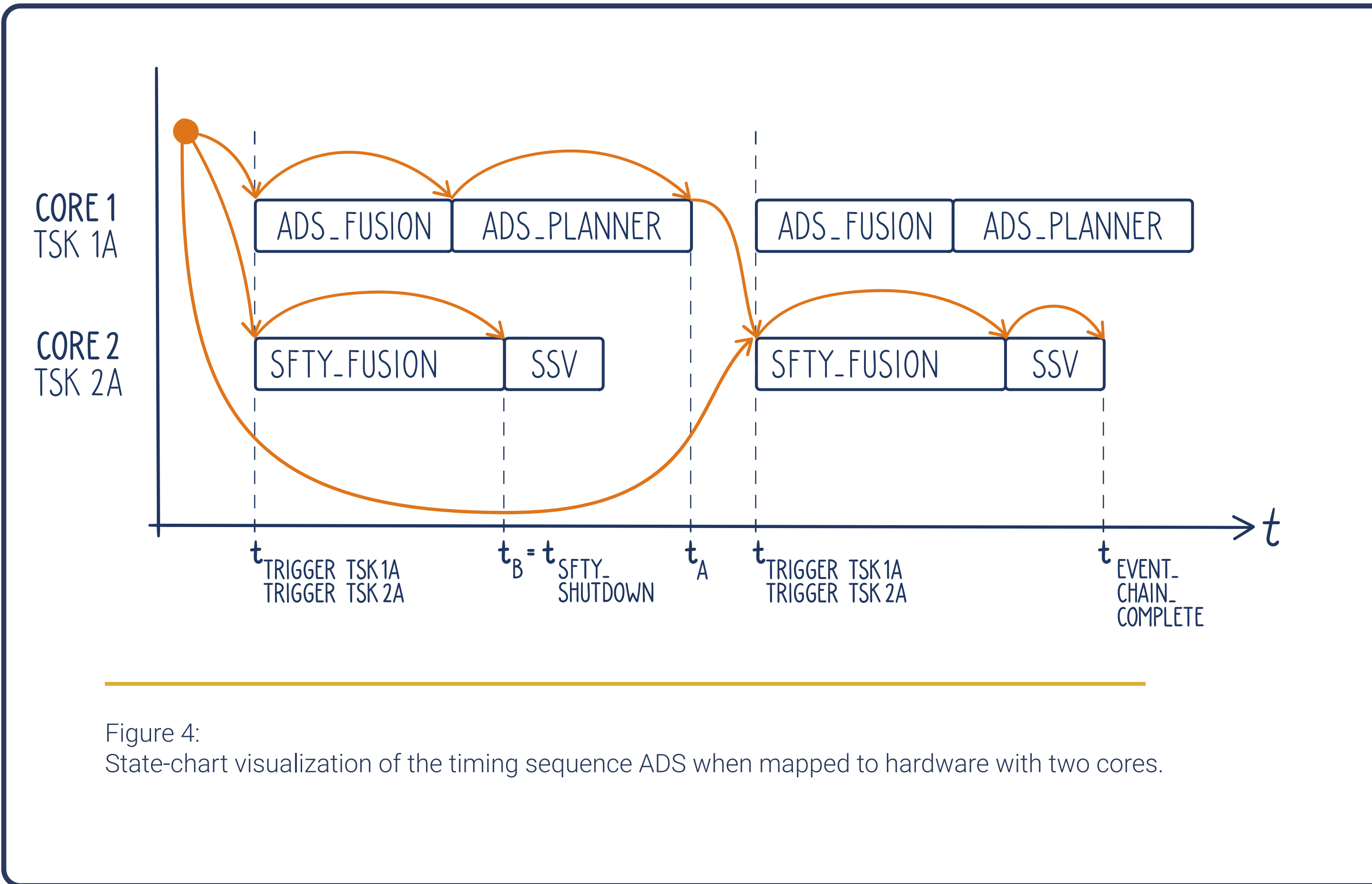


Figure 4:
State-chart visualization of the timing sequence ADS when mapped to hardware with two cores.

Including the timing of event chains within the requirements management can resolve such challenges independently of further architectural decisions, such as mapping functions to processes on cores. With event chains providing the exact timing requirements, the system integrator clearly sees the synchronicity requirements' violation (see Figure 4). They can now apply countermeasures without having to involve the function developers.

Modularity

The future of the software-defined vehicle depends on modularity. New functions can be added to existing systems that rely on data provided by existing sensors and data-fusion modules. The challenge is proving that adding these capabilities does not impact existing functionality.

Let's assume that our simple ADS will be extended to include a lane change maneuver (LCM) function. Because of the increased complexity, several functional requirements must be added to the ADS_Planner. All this information is captured in a new event chain, "LCM." The new system now has to implement both the ADS event chain and the new LCM event chain (refer to Figure 3). It should be noted that the LCM event chain would structurally look the same but with more functional requirements attached to it. It should be kept in mind that each event chain refers to different scenarios. ADS addresses driving scenarios where no lane change is involved, and LCM addresses scenarios for the extended functionality, including lane change. The effect on the existing implementation is that more functional requirements need to be implemented by ADS_-Planner, resulting in longer overall runtimes.

Let's further assume our vehicle OEM has two car models, a base car with simple hardware named 'Small-Foot,' and a luxury car with more powerful hardware named 'Big-Foot.'

The new implementation of ADS_Planner misses its timing requirements when running on the Small-Foot hardware but meets them when executed on Big-Foot. Therefore, the OEM can offer the upgrade only for luxury cars by exchanging the implementation of ADS_Planner.

Using event chains can even help when more detail is needed. Let's assume the new implementation of ADS_Planner would miss its timing requirements even on Big-Foot but only under specific environmental conditions. This occurs when the vehicle is inside a tunnel with many fluorescent tubes, making the vision process more computationally intensive. In this case, the developers could decide to disable lane-change at runtime if vision processing exceeds a specific timing budget and temporarily fall back to ADS without lane-change. This detailed understanding of event chains helps maintain traceability between functional and timing requirements and their implementation, even in complex scenarios determined by hardware decisions, variants, and other choices.

Test and Verification

Testing traditionally focuses on correct functionality, with tests designed to check that, for example, the headlights turn on through manual switch operation or in response to sundown while driving.

What is overlooked, however, is the temporal aspect of testing. For example, does our ADS_-Planner step still fulfill its timing requirements once integrated alongside other functions? With timing as part of the requirements, it is easy to create such tests and ensure that they are tested throughout the development process and under the defined operational conditions. Event chains allow a clear definition of a start-event, and identification of intermediate steps and phrase timing requirements between those steps. Event chain steps map to functions within the function net. Therefore, events emitted by these functions (e.g., start or end of a function) can now be seen in the context of the state of the event chain – the event chain instance. The event chain can be seen as a specification of what an observer needs to observe to check whether the device or system under test meets its requirements.

Any time the start event of an event chain becomes observable in the system under test, the observer would start a new instance of the event chain and assign a state to it. The state describes how far the instance could have progressed due to the events observed after the start event. The observer can therefore check any requirements based on the steps currently being progressed or switched on.

The example in Figure 4 showed how an observer would trigger the yellow event chain instance while observing the starts and ends of the functions (any yellow arrow marks a new state) referenced from event chain ADS in Figure 3. As soon as a test bench is available to observe the steps relevant to the event chains, the above concept of an observer based on event chains can easily verify even complex timing requirements such as the one depicted earlier related to synchronization. This is the case even if one event chain is simultaneously active in multiple event chain instances because its start event has been triggered multiple times.

Note that once all timing requirements are captured, and all timing parameters needed to approximate the timing behavior of the system under test are available, it is possible to verify the timing requirements. This is achieved even in non-functional simulations with tools like chronSUITE, even before a functional implementation is available. Such simulations are helpful to assure fulfillment of timing requirements after development iterations, feature enhancements, and other changes without having to go through HIL testing. It also allows consistency checking and a review of the static architecture against the dynamic requirements at the earlier stages of the development process.

Summary

As the automotive industry transitions to the software-defined vehicle, domain controllers, and high-performance computing (HPC), it is clear that the traditional approach to software development is no longer suitable. With a focus on modularity, software reuse, and over-the-air updates for fixes and upgrades, the methods used to create yesterday's static architectures no longer work for dynamic, AUTOSAR Adaptive solutions.

As highlighted through the examples provided here, specifying timing requirements early and in the context of scenarios separates the dynamic architecture from the static architecture, even at functional level. This leaves architects and system integrators with more degrees of freedom when undertaking design decisions. As they work through hardware selection or the assignment of tasks to core, the demands of each task or runnable in the context of the broader application are significantly clearer.

Additionally, it delivers greater stability at the logical layer. Bearing in mind the importance of reuse, timing requirements can live with their functions and scenarios, allowing code to be reused more successfully in new projects or even in cost-optimized solutions. Thanks to the derived timing tests and verification, it also provides more confidence when new functionality is added to an existing codebase.

Event-chain-centered architecture design allows system architects and automotive development teams to tackle these new challenges. Activity charts, as specified by the OMG, are the ideal technology to describe event-chains simply and effectively.

Automotive software teams that move to a dynamic, event-chain-driven system design process find that the timing-related issues described here rarely appear in their projects. After hundreds of projects at a host of automotive OEMs, Tier 1s, and sub-suppliers, Inchron's chronSUITE has not only become a standard tool in the development process, it is recognized as important enough to warrant changing that process.

If you'd like to learn more about how Inchron's chronSUITE is used to tackle such challenges, whether on a running project or from the start of a new one, please feel free to get in touch.

We'd be more than happy to help.

References

1 O. Burrkacky et al., “When code is king: Mastering automotive software excellence,” McKinsey & Company, February 2021: https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/when-code-is-king-mastering-automotive-software-excellence

2 ”OMG Systems Modeling Language (OMG SysML™),” The Object Management Group, November 2019: https://www.omg.org/spec/SysML/1.6/PDF, Chapter 11.2.1ff

3 M. Törngren et al., “Architecting Safety Supervisors for High Levels of Automated Driving,” 2018 21st International Conference on Intelligent Transportation Systems (ITSC), Maui, HI, 2018, pp. 1721-1728. doi: 10.1109/ITSC.2018.8569945