Received 10 May 2022, accepted 17 July 2022, date of publication 25 July 2022, date of current version 3 August 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3193495

RESEARCH ARTICLE

Integrated System Design and Safety Framework for Model-Based Safety Assessment

RAHUL KRISHNAN, (Member, IEEE),
AND SHAMSNAZ VIRANI BHADA, (Senior Member, IEEE)
Department of Electrical and Computer Engineering, Worcester Polytechnic Institute (WPI), Worcester, MA 01609, USA
Corresponding author: Rahul Krishnan (rkrishnan2@wpi.edu)

ABSTRACT The increased complexity of modern engineered systems has introduced novel challenges for assessing their safety early in the life cycle. For example, due to the iterative nature of the design and safety life cycle, there is constant data transformation and feedback of information between the system design models, safety analyses, and safety verification. Data transformation and feedback are often manually performed by engineers, which is time-consuming and error prone and can introduce inconsistencies in safety assessments. Although several model-based systems engineering approaches have been developed for safety analysis and safety verification, current approaches do not address the inconsistencies introduced in the safety assessment process. This study describes the Integrated System Design and Safety (ISDS) framework, which is a model-based safety assessment framework that aims to eliminate such inconsistencies. The framework combines a model-based safety analysis approach with a model-based safety verification. This paper extends previous work, which focused on the model-based safety analysis approach, to describe the model-based safety verification approach adopted in the ISDS framework. Safety verification is performed using a simulation-based fault injection approach and enabled by a fault injection engine, which injects failures into the system design and characterizes system behaviors to identify safety violations impacting the system. The results from the case study, in which the framework is used to assess the safety of a forward collision warning system, highlight that the algorithms and automated feedback loops of the framework can reduce inconsistencies in the safety assessment process while also identifying safety violations impacting the system.

INDEX TERMS Model-based systems engineering (MBSE), safety analysis, failure modes and effects analysis (FMEA), systems engineering, SysML, simulation-based fault injection, safety verification.

I. INTRODUCTION

Technological advances in recent years have led to the design of highly complex and sophisticated systems, such as autonomous vehicles, robots, medical devices, and even financial-trading systems, that can significantly improve our quality of life [1]. The scale of adoption of such systems is evident based on future market trends. The market size for autonomous vehicles is projected to reach 60 billion US dollars by 2030 [2], and the intelligent robot market is projected to reach 3 billion US dollars by 2026 [3]. While these complex systems have numerous benefits, they also introduce new challenges [1]. One of the core challenges lies in assessing the safety of such complex systems [4], [5].

The safety assessment of a system is a judgment made about the safety conformance and safety integrity achieved by every safety instrumented function within the system [6]. This judgment is based on a) the safety of the system development process (i.e., is the design process mature and in conformance with the criteria stated in the safety standards?) and b) the safety of the system design (i.e., does the system design achieve the required safety integrity level?). While both factors are important for completing a safety assessment, this research will only focus on the latter (i.e., safety assessment based on the safety of the system design).

The associate editor coordinating the review of this manuscript and approving it for publication was Chaitanya U. Kshirsagar.

Academic researchers, industry experts and safety standards recommend assessing the safety of the system design using a combination of safety analyses, safety verification, and testing at various stages in the life cycle [6]–[9]. However, identifying safety-related design issues early in the life cycle can reduce rework, costs, and schedule delays [9]–[11]. While testing occurs in the later stages of the life cycle, safety analyses and safety verification can be performed early in the life cycle on the available design models and offer potential solutions for the early identification of safety-related design issues.

Safety analysis is performed using a combination of one or more techniques, such as failure modes and effects analysis (FMEA), fault tree analysis (FTA), and hazard analysis [7], [8]. These analyses are often performed using independent tools [12]. Each tool works off its own system design model; engineers must manually extract the relevant information from the original model and either import, or worse, re-create the system design model in each tool [12], [13]. Not only is this activity time-consuming and error prone, but the existence of multiple system design models also creates a lack of traceability between the models and safety analyses [14]. Another limitation is that the failure to update the system design model in each tool may create an inconsistency between the current design model and the safety analyses, leading to an incorrect safety assessment [12], [14], [15].

Similar limitations also plague the activities performed during safety verification. In formal methods, a common technique for safety verification early in the life cycle [16], specific model checkers are used to perform safety verification. The model checking tool can use one of the many available languages to describe the system design. Performing safety verification requires a description or transformation of the original system design to the specific language used by the model checker [17]. This transformation may create an inconsistency between the current system design and safety verification, consequently lead to an incorrect safety assessment. The main objective of this paper is to eliminate such manual activities that are performed during safety assessment as well as the inconsistencies they introduce.

To address these limitations, researchers have adopted a model-based systems engineering (MBSE) approach to safety assessment [12], [18], [19]. Studies have shown that adapting an MBSE approach to system development, commonly referred to as model-based development (MBD), improves the completeness and consistency in system development [20], fosters improved communication across design teams [20], provides added traceability between different models of the system [21], and enables easy integration with other engineering analysis tools [22]. The increased adoption of MBD also enhances the ability to perform safety analyses and safety verification early in the life cycle [23]. Consequently, an MBD approach to safety assessment offers a solution for resolving inconsistencies that arise in the safety assessment process. MBD also enables verification activities such as formal methods and simulation testing using early system design models, which helps to characterize and observe design issues in the context of safety assessments. This is because MBD allows a more formal specification of a system's intended behavior with respect to its requirements and enables the integration of a variety of analyses or simulations to be performed on the available system design model, such as formal methods or fault injection [16], [24]–[26]. However, there is currently no framework or method that allows for feedback from safety analyses or safety verification to the system design model while also identifying safety-related design issues impacting the system. The ISDS framework described in this paper aims to address this gap by leveraging MBD to automate the feedback between the system design model, safety analyses, and safety verification.

The authors' previous work [27] described the model-based safety analysis approach used in the ISDS framework (left side of the framework). The paper highlighted how feedback loops between the SysML model and safety analyses (such as FMEA and FTA) are used not only to automatically generate FMEA tables and fault trees from the SysML model but also to automatically update the SysML model with any changes made to the safety artifacts. By eliminating the manual tasks performed by engineers, the framework could reduce the inconsistencies introduced in the safety assessment process. This paper extends the previous work and describes the model-based safety verification approach used in the ISDS framework. The remainder of the paper is organized as follows. Section II discusses the relevant literature in the field of safety verification. Section III describes the approach used for safety verification in the ISDS framework. Section IV demonstrates the application of the framework to the design of a forward collision warning system case study. Section V concludes the paper.

II. LITERATURE REVIEW

Safety verification is performed on the right side of the Vee development methodology [28] and is used to determine if the system design meets the safety requirements [6]. Verification techniques such as formal methods and fault injection have been extensively used in MBD for safety verification of complex systems [16]. Formal methods rely on mathematically proving that a design satisfies a set of defined safety properties. A popular formal methods technique is model checking, which is used to verify finite-state systems [29]. The model checking technique is defined as follows: given a model of a system for which the set of finite state transitions has been encoded (i.e., system model) and a set of safety properties that the system must satisfy (i.e., safety requirements), the model checking problem is to identify all states in the system model that satisfy the requirements. Table 1 summarizes the commonly used model checking tools, such as the safety-critical application development environment (SCADE) design verifier [30] and NuSMV model checker [31]. In [24], [26], the authors developed the system design model in Simulink and used the SCADE

TABLE 1. **Summary of model checking approaches used for safety verification.**

Authors	Modeling language	Model checking tool
Joshi, Heimdahl [24]	Simulink	SCADE design verifier
Wang et al. [26]	Simulink	SCADE design verifier
Mhenni et al. [12]	SysML	NuSMV
Wang et al. [17]	SysML	NuSMV

TABLE 2. **Summary of simulation-based fault injection approaches used for safety verification.**

Authors	Modeling language	Fault injection tool
Jha et al. [33]	No model consideration	AVFI
Jha et al. [34]	No model consideration	Kayotee
Li et al. [35]	No model consideration	AV-Fuzzer
Juez et al. [36]	Simulink	Sabotage
Juez et al. [19]	SysML/RobotML	FI framework

design verifier to automate safety verification via model checking. In [12], [17], the authors developed the system design model in SysML and used the NuSMV symbolic model checker to perform safety verification. Overall, model checking has been shown to allow the verification of system safety properties based on requirements using automated verification procedures [25]. However, the limitation with model checking is that it relies heavily on the knowledge and experience of the engineer to define the safety properties and system states [32], which can be challenging for complex systems that operate in diverse environments. Another challenge (and ongoing research problem) with model checking is the state explosion problem, where the increase in system complexity leads to a state space that is too large for the technique to be computationally viable [29].

To overcome some of the model checking challenges, researchers have used fault injection for safety verification of complex systems [37]. The fault injection technique is effective at analyzing faulty system behavior in a controlled environment by forcing faults and other exceptional conditions and observing the resulting system behavior [32]. Additionally, fault injection can reduce reliance on the expertise of engineers because it allows for automation when injecting faults into the system design model. The automatic nature of verification makes the high level of usability attractive to nonexperts in the verification process as well [25].

Fault injection techniques are broadly classified into two approaches: hardware-based and software-based approaches. Hardware-based approaches are accomplished at the physical level and use external sources to introduce faults into the system's hardware [38], while software-based (also referred to as simulation-based) approaches mimic faults through code changes or by injecting the effects of faults into the code [39]. The advantages of simulation-based approaches are that they pose no risk of damage to the system, require fewer resources to execute in terms of time and effort, and have higher observability (how well the effects of faults on the system can be measured) and controllability (how well the location of faults in space and time can be controlled) of system behavior in the presence of faults [40], [41]. This approach is particularly suited for an MBD environment; it can provide timely feedback on the safety verification of the system design model early in the life cycle [38]. Additionally, high-fidelity simulators allow engineers to detect gaps in requirements, identify new test cases or observe unforeseen system behavior caused by the injected faults in realistic operational environments earlier in the life cycle [37], [42], [43]. Table 2 summarizes the simulation-based fault injection approaches that have been developed for safety verification.

Jha *et al.* [33] developed a fault injection tool called an autonomous vehicle fault injector (AVFI) that uses the Car Learn and Act (CARLA) simulator [44] for resilience assessment of autonomous vehicle systems. The tool can inject hardware, data, timing, and machine learning-related faults (source level fault models) using source code instrumentation at run time and compute resilience metrics such as mission success rate, traffic violations per kilometer, accidents per kilometer and time to traffic violations. Jha *et al.* [34] also developed the Kayotee fault injection tool, which uses the drivesim simulator [45] for safety and reliability assessment. Along with source level faults, the tool can inject faults modeled as bit-flips in different functional units of the processor to identify unforeseen behavior under faulty conditions, such as safety envelope breaches and lane-centering breaches for autonomous vehicles. Li *et al.* [35] developed the AV-Fuzzer fault injection tool, which uses the Silicon Valley Lab (SVL) simulator [46] to find safety violations in autonomous vehicles under evolving traffic conditions. The tool uses a genetic algorithm to identify new operational scenarios or test cases that could lead to violations of requirements that are modeled as safety constraints. While such tools have shown success at identifying safety-related issues caused by faulty system behavior, they are not sufficient to perform a safety assessment since they do not consider the system design model as an input to safety verification. To leverage the benefits of MBD for early safety assessments, safety verification must be integrated with the design and safety life cycle (i.e., the safety verification process should be based on the available system design model as well as the safety analyses that were performed on that design model).

Juez *et al.* [19], [36] overcome this problem by combining a model-based design process with a simulation-based fault injection technique. In [36], the authors use Simulink models to represent the system design and the Sabotage fault injection framework to perform safety verification early in the life cycle. The Sabotage framework uses the Simulink model to configure the type, location, and time for the faults to be injected and the Dynacar simulator [47] to perform fault injection simulation. The framework uses source code instrumentation to inject faults and calculates the effect of the faults by computing the fault tolerant time interval (FTTI)

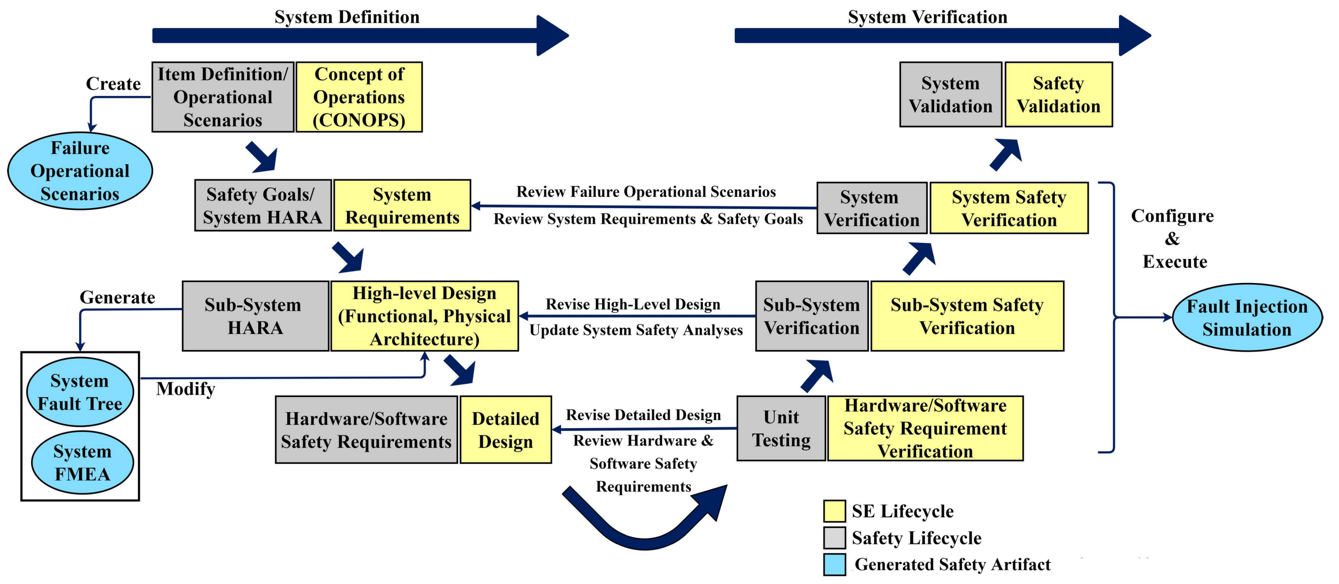

System Definition
System Verification
Create
Item Definition/ Operational Scenarios
Concept of Operations (CONOPS)
Failure Operational Scenarios
System Validation
Safety Validation
Safety Goals/ System HARA
System Requirements
Review Failure Operational Scenarios
Review System Requirements & Safety Goals
System Verification
System Safety Verification
Configure & Execute
Generate
Sub-System HARA
High-level Design (Functional, Physical Architecture)
Revise High-Level Design
Update System Safety Analyses
Sub-System Verification
Sub-System Safety Verification
Fault Injection Simulation
System Fault Tree
System FMEA
Modify
Hardware/Software Safety Requirements
Detailed Design
Revise Detailed Design
Review Hardware & Software Safety Requirements
Unit Testing
Hardware/Software Safety Requirement Verification
SE Lifecycle
Safety Lifecycle
Generated Safety Artifact

FIGURE 1. ISDS framework uses a model-based system engineering approach for the safety assessment of complex systems. The left side of the framework corresponds to the system definition phase of the life cycle, where different design and safety information is captured using different SysML diagrams but a single SysML model. Here, the framework focuses on performing safety analyses on system design and safety models (using FMEA and FTA techniques) as part of the safety assessment. The right side of the framework corresponds to the verification phase of the life cycle, where the system design is verified against safety requirements at the component, subsystem, and system levels. Here, the framework focuses on performing safety verification on the system design and safety models (using simulation-based fault injection techniques) as part of the safety assessment.

(i.e., the time interval between the injection of the fault and the observation of the fault's system level effect). The limitations of this approach arise from the Simulink models, which are good for mathematically representing the system design but cannot be used to store other design and safety data, such as requirements, use cases, failure modes, etc. Additionally, the framework does not use the results from the safety analyses to configure the fault injection. This omission can cause an inconsistency between the results of the safety analyses performed during system definition (left side of the Vee development process) and the safety verification (right side of the Vee development process).

The authors overcome these limitations in [19] by developing system design models in SysML. The general-purpose nature of SysML allows different design and safety data to be stored in the same model. The approach uses the SysML model and the results from the FMEA to configure the fault injection in a robot and environment simulator called Gazebo [48]. Using source code instrumentation, different failure modes were injected to simulate the system design in the presence of faults. The resulting behavior was evaluated against a predefined component level safety requirement stored in the SysML model. While this approach presents a promising solution for early safety assessments of complex systems in an MBD environment, it assumes a highly linear approach toward safety assessments. In practice, the design and safety life cycle are iterative, with feedback loops between the system design and safety analyses and the system design and safety verification. Fig. 2 illustrates this concept. Additionally, the lack of feedback from safety verification

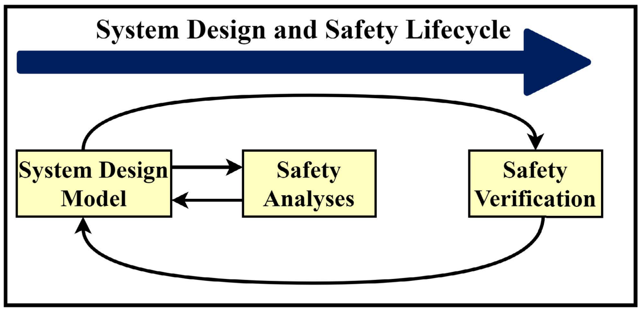


FIGURE 2. Feedback loops that exist between the system design model, safety analyses, and safety verification, which highlight the iterative nature of the design and safety life cycle.

to the system design model can introduce inconsistencies in later stages of the life cycle since the results from the verification are not captured in the SysML model.

The ISDS framework addresses these problems by integrating the system design and safety life cycle and incorporating feedback between the system design models, safety analyses, and safety verification. The framework uses SysML to model the system architecture and define the system design and safety data. There are several other candidate modeling languages that can be used to implement MBD, such as Modelica [49]. Modelica is an object-oriented language for describing differential algebraic equation (DAE) systems combined with discrete events. It is an expressive formal language and its DAE solving capabilities can support various analyses. However, the strength of SysML lies in its

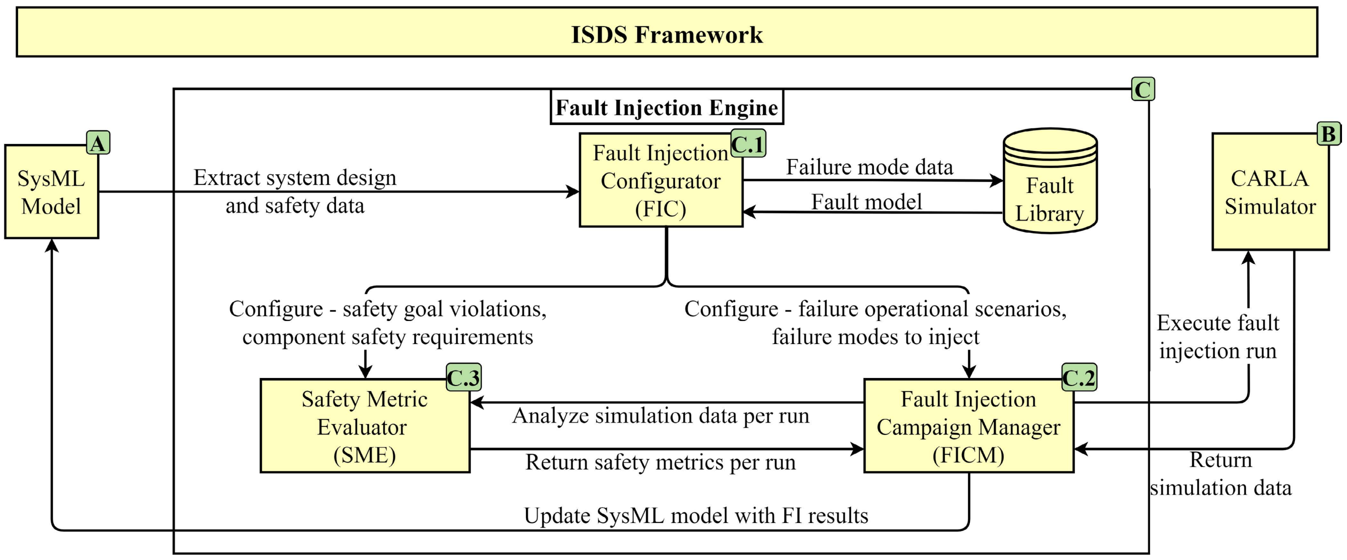

ISDS Framework
Fault Injection Engine
C
A
SysML Model
Extract system design and safety data
Fault Injection Configurator (FIC)
C.1
Failure mode data
Fault model
Fault Library
B
CARLA Simulator
Configure - safety goal violations, component safety requirements
Configure - failure operational scenarios, failure modes to inject
Execute fault injection run
C.3
Safety Metric Evaluator (SME)
Analyze simulation data per run
Return safety metrics per run
C.2
Fault Injection Campaign Manager (FICM)
Return simulation data
Update SysML model with FI results

FIGURE 3. **Overview of the fault injection engine used for safety verification in the ISDS framework.**

descriptive modeling capabilities, with expressive constructs for requirements, structural decomposition, logical behavior, and cross-cutting constructs [50]. Additionally, SysML is the preferred language to model the system architecture among system engineers [22] and has become the de facto modeling language in systems engineering [51]–[53]. For these reasons, SysML was the preferred modeling language to implement MBD in the ISDS framework.

The authors' previous work [27] highlighted how the model-based safety analysis approach of the ISDS framework reduces the inconsistencies introduced in the safety assessment process by incorporating automated feedback loops between the SysML model and safety analyses. This paper extends previous work and describes the model-based safety verification approach of the ISDS framework. The approach aims to overcome the limitations of the current literature by incorporating automated feedback loops between safety verification and the system design model. The feedback loops help in reducing the inconsistencies introduced in the safety assessment process while also verifying the design by identifying safety violations or safety-related design issues impacting the system. The next section provides a detailed description of the methodology for safety verification in the ISDS framework.

III. ISDS FRAMEWORK: APPROACH FOR SAFETY VERIFICATION

This section provides a brief overview of the ISDS framework followed by a detailed description of the approach for safety verification. The ISDS framework (see Fig. 1) adopts the Vee development methodology and an MBD approach to integrate the system design and the safety life cycle by combining data into a single SysML model. The left side of the framework focuses on system definition, where system design and safety data are incrementally added to the model. Once the high-level architecture models (functional and logical architecture) and subsystem hazard and risk assessments (HARA) are complete, the model is exported as an XML file, and Python scripts are used to automatically generate system-level fault trees and FMEA tables. Any changes made (after review by an engineer) are updated in the XML file and subsequently the SysML model. Finally, component (hardware or software) safety requirements are added, if applicable, for the failure modes. The right side of the framework focuses on verifying that the system design satisfies the safety requirements for all the identified operational scenarios. The verification process is based on simulation-based fault injection, which allows for the observation of safety requirement violations even in the presence of faults. The next section describes the methodology for safety verification using fault injection– how the SysML model is used to configure the fault injection simulation, the process of running the simulation with faults injected, the computation of safety metrics based on the safety requirements, and how the SysML model is updated with the results of the safety verification.

The safety verification approach in the ISDS framework, shown in Fig. 3, consists of three components: a) the SysML model containing system design and safety data developed during the system definition; b) the CARLA simulator [44], which is a high-fidelity simulator for recreating autonomous vehicle behavior in realistic operational environments; and c) the fault injection engine used to configure, run, and analyze the simulation runs to compute safety metrics. The fault injection engine acts as a bridge between the SysML model and the CARLA simulator. It extracts the system design and safety data from the SysML model, configures the CARLA simulation based on the extracted data, executes

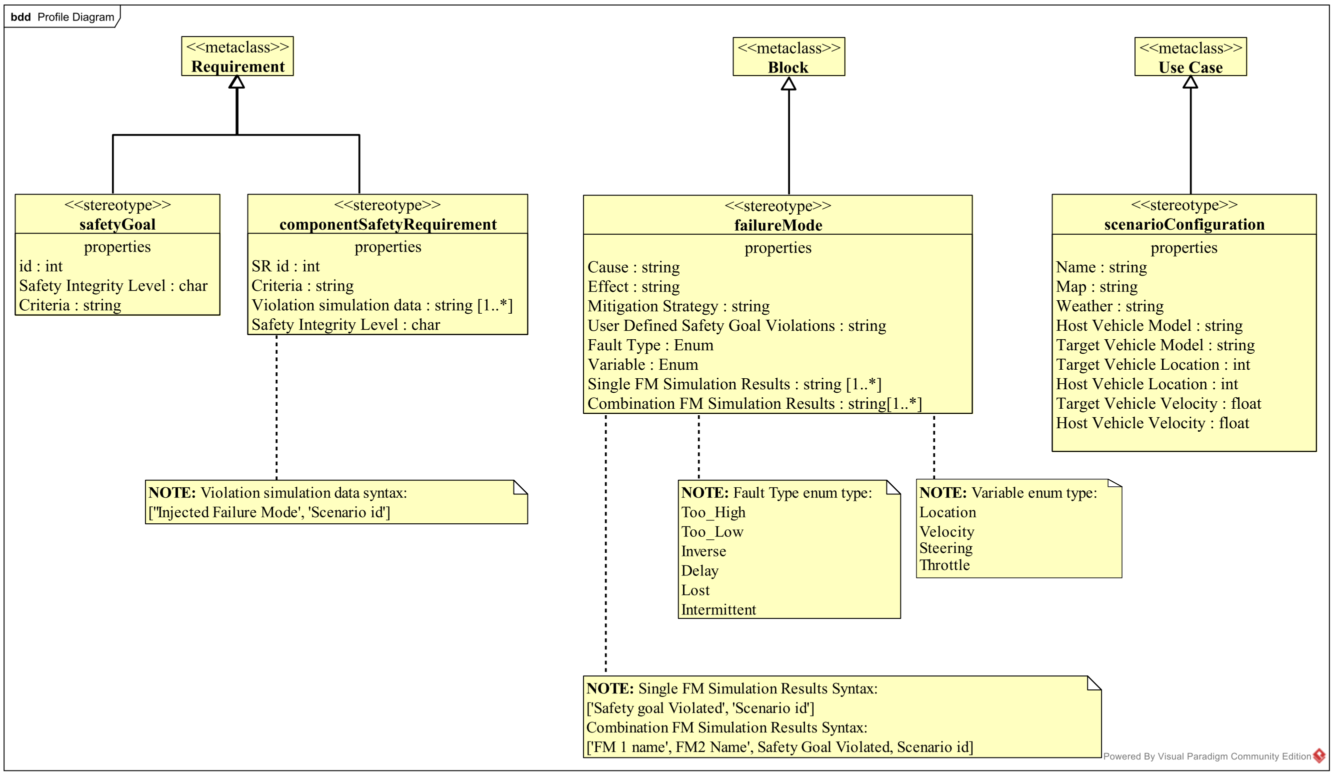

bdd Profile Diagram
<<metaclass>> Requirement
<<metaclass>> Block
<<metaclass>> Use Case
<<stereotype>> safetyGoal
properties
id : int
Safety Integrity Level : char
Criteria : string
<<stereotype>> componentSafetyRequirement
properties
SR id : int
Criteria : string
Violation simulation data : string [1..*]
Safety Integrity Level : char
<<stereotype>> failureMode
properties
Cause : string
Effect : string
Mitigation Strategy : string
User Defined Safety Goal Violations : string
Fault Type : Enum
Variable : Enum
Single FM Simulation Results : string [1..*]
Combination FM Simulation Results : string[1..*]
<<stereotype>> scenarioConfiguration
properties
Name : string
Map : string
Weather : string
Host Vehicle Model : string
Target Vehicle Model : string
Target Vehicle Location : int
Host Vehicle Location : int
Target Vehicle Velocity : float
Host Vehicle Velocity : float
NOTE: Violation simulation data syntax: ['Injected Failure Mode', 'Scenario id']
NOTE: Fault Type enum type: Too_High Too_Low Inverse Delay Lost Intermittent
NOTE: Variable enum type: Location Velocity Steering Throttle
NOTE: Single FM Simulation Results Syntax: ['Safety goal Violated', 'Scenario id'] Combination FM Simulation Results Syntax: ['FM 1 name', FM2 Name', Safety Goal Violated, Scenario id]
Powered By Visual Paradigm Community Edition

FIGURE 4. **Safety profile of the ISDS framework. The safety profile illustrates how native SysML elements are extended using stereotypes to represent safety-related information in the SysML model. The triangle from the parent element to the child element represents an extension or inheritance relationship and can be read as "is a type of". The child element contains additional properties to capture safety-related information. The component safety requirement and failure mode stereotype contain tagged values that are placeholders for the results from the safety verification. The syntax for the respective tagged values is shown in the figure. Additionally, the six fault types and four variables for which fault injection is supported are also shown in the figure.**

fault injection, verifies the system design against the safety requirements, and updates the SysML model with results from fault injection. The next section describes the three components of the safety verification approach in the ISDS framework in greater detail.

A. SysML MODEL

The description provided in this section assumes that the SysML model was developed in accordance with the system definition phase of the ISDS framework [27]. Instead of re-iterating the activities in the system definition phase of the framework, this section summarizes the SysML model elements that influence the safety verification approach of the ISDS framework.

The SysML model contains the design and safety data developed during the system definition (left side of the Vee) of the ISDS framework. The model consists of multiple diagrams, each representing a different view of the system and containing specific design and safety data, as shown in Fig. 5. The framework uses SysML profiles, called safety profiles, to store design and safety data in the same SysML model. Fig. 4 illustrates the safety profile of the ISDS framework. The safety profile is a SysML profile used in the ISDS framework that allows data from different domains to be stored in the same SysML model, i.e., system design data, safety data, and simulation-specific properties. Native SysML elements are extended (i.e., customized) for storing specific types of data based on the domain and platform of the modeled system. Stereotypes and tagged values are used to extend the reference meta-class of a native SysML element.

The requirement meta-class is extended to create the safety goal and component safety requirement stereotype. The safety goal stereotype contains an identifier, a safety integrity level (or risk), and the criteria that represent the safety goal violations. Similarly, the component safety requirement stereotype contains an identifier, the criteria that represent the safety requirement violations, and a variable to store the results from the fault injection simulation, called the violation simulation data. The action meta-class is extended to create the failure mode stereotype. Each failure mode contains a cause, an effect, a mitigation strategy, user-defined safety goal violations (each is string data type), a fault model, a variable (each is an enum data type) and a variable to store the results from the fault injection simulation (one for a single failure mode injection and one for the injection of multiple failure modes). Fig. 4 shows the syntax in which the simulation results are stored in the SysML model. Finally, the use case meta-class is extended to create the

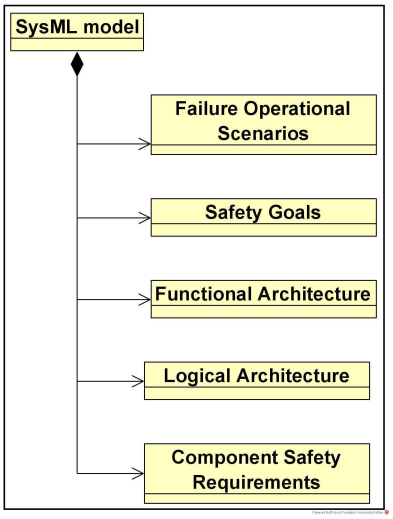
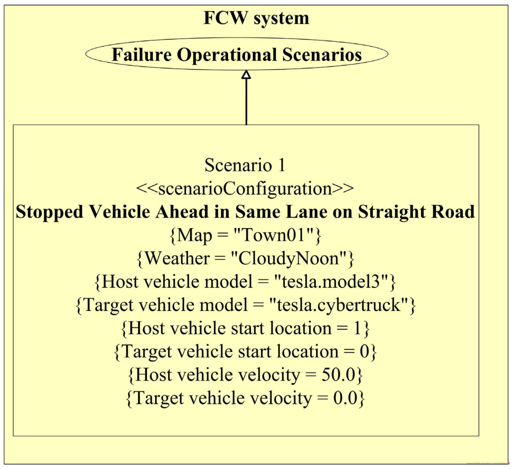
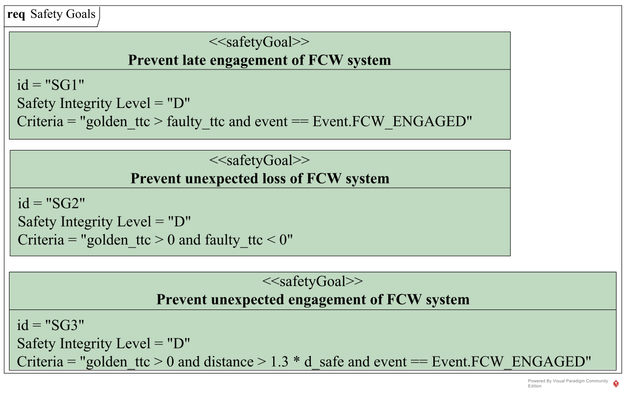


FIGURE 5. Overview of the system design and safety data stored in the SysML Model. The filled diamond represents the "is made up of" relationship in SysML.



Scenario 1
<<scenarioConfiguration>>
Stopped Vehicle Ahead in Same Lane on Straight Road
{Map = "Town01"}
{Weather = "CloudyNoon"}
{Host vehicle model = "tesla.model3"}
{Target vehicle model = "tesla.cybertruck"}
{Host vehicle start location = 1}
{Target vehicle start location = 0}
{Host vehicle velocity = 50.0}
{Target vehicle velocity = 0.0}

FIGURE 6. Failure operational scenario for the safety assessment of a vehicle's forward collision warning system. The scenario is captured in a use case diagram and contains the configurable parameters to setup up the fault injection simulation. These parameters capture the environmental variables and their states.



<<safetyGoal>>
Prevent late engagement of FCW system
id = "SG1"
Safety Integrity Level = "D"
Criteria = "golden_ttc > faulty_ttc and event == Event.FCW_ENGAGED"

<<safetyGoal>>
Prevent unexpected loss of FCW system
id = "SG2"
Safety Integrity Level = "D"
Criteria = "golden_ttc > 0 and faulty_ttc < 0"

<<safetyGoal>>
Prevent unexpected engagement of FCW system
id = "SG3"
Safety Integrity Level = "D"
Criteria = "golden_ttc > 0 and distance > 1.3 * d_safe and event == Event.FCW_ENGAGED"

FIGURE 7. Safety Goals of a forward collision warning system. The safety goals have been identified using the HAZOP method and represent the high-level safety requirements of the system.

'scenarioConfiguration' stereotype, which contains variables to configure the operational environment in the simulation. The next section describes the different views of the SysML model, as shown in Fig. 5, in detail.

1) FAILURE OPERATIONAL SCENARIOS

Failure operational scenarios specify the environment and scenario under which fault injection is to be performed. The scenarios are defined by use case diagrams in the SysML model. This diagram is used to configure keywords, environment variables, and variable states that will set up the failure operational scenario in CARLA. The definitions that are specified for the operational scenario and supported by the simulator include the following: the road type, weather conditions, host vehicle data (vehicle type, starting location, and desired speed), and nonhost vehicle data (vehicle type, starting location, desired speed, and specific behaviors). These definitions are captured as tagged values under the ≪scenarioConfiguration≫ stereotype, as shown in Fig. 6. The failure operational scenario provide a basic representation for recreating scenarios in simulation. They do not provide any support for uncertainty or perturbances in scenario parameters.

2) SAFETY GOALS

Safety goals represent the system level safety requirements and are defined by a requirements diagram in the SysML model. This diagram is used to configure the safety goal metric in the simulator, i.e., the metric that evaluates if a safety goal was violated. Each safety goal defined in the diagram contains a safety goal ID, the safety integrity level (or associated risk) of the safety goal, and the evaluation criteria that represent the safety goal violations. These definitions are captured as tagged values under the ≪safetyGoal≫ stereotype.

3) FUNCTIONAL ARCHITECTURE

The functional architecture contains the different system functions and their interactions and is defined by an activity diagram in SysML (Fig. 8). Each function in this activity diagram is linked to another activity diagram that contains the function-specific safety information–the functional failure modes, causes, effects, user-defined safety goal violations, risk level, mitigation strategy, and simulation results. Additionally, it contains a variable and fault model, which is used to characterize the failure mode in the simulation. The safety-specific information is extracted from the SysML model to generate the FMEA. The activity diagrams that make up the functional architecture are used to configure the failure modes to be injected in the simulation. The failure modes of a function are captured under the ≪failureMode≫

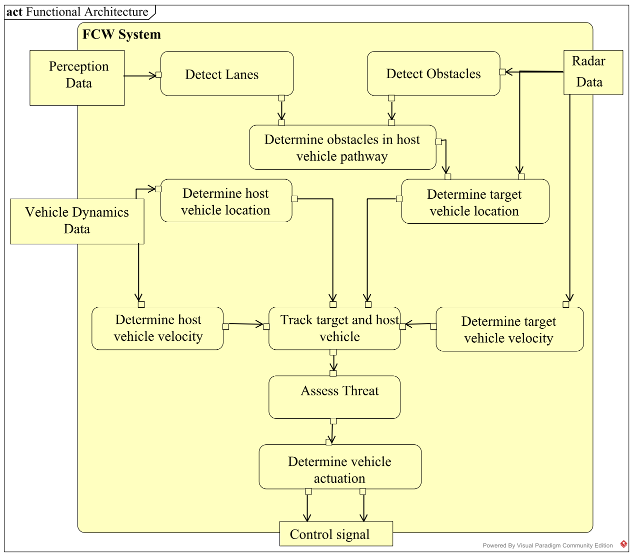

act Functional Architecture
FCW System
Perception Data
Detect Lanes
Detect Obstacles
Radar Data
Determine obstacles in host vehicle pathway
Vehicle Dynamics Data
Determine host vehicle location
Determine target vehicle location
Determine host vehicle velocity
Track target and host vehicle
Determine target vehicle velocity
Assess Threat
Determine vehicle actuation
Control signal
Powered By Visual Paradigm Community Edition

FIGURE 8. Functional architecture of a forward collision warning system. The functional architecture contains the different functions of the system as well as the flow of information between the functions. Each function is linked to a subdiagram that contains the failure modes of the function.

stereotype, and the safety information is captured as tagged values under the ≪failureMode≫ stereotype (see Fig. 9).

4) LOGICAL ARCHITECTURE

The logical architecture represents the system's structural design and is defined by the block definition diagram (BDD) and internal block diagram (IBD) (Fig. 10 and Fig. 11, respectively). The BDD contains logical blocks that represent subsystems. Functions from the functional architecture diagram are allocated to corresponding blocks in the logical architecture. The BDD is used to organize the fault injection simulations such that only the failure modes of functions allocated to the specific block are injected in the simulation. The IBD captures the interconnections and data flows between the blocks modeled in the BDD.

5) COMPONENT SAFETY REQUIREMENTS

Component safety requirements represent hardware or software safety requirements allocated to the blocks in the logical architecture and are defined by a requirements diagram in SysML. To ensure traceability, the requirements diagram is linked to the corresponding block in the logical architecture as a subdiagram. Each component safety requirement defined in the diagram contains an identifier, the evaluation criteria that represent the requirement violation, and a variable to store the results from the fault injection, called violation simulation data. The definitions are captured as tagged values under the ≪componentSafetyRequirement≫ stereotype (see Fig. 12).

B. CARLA SIMULATOR

CARLA is an open-source simulator for autonomous driving systems. It provides support for a variety of sensor suites,

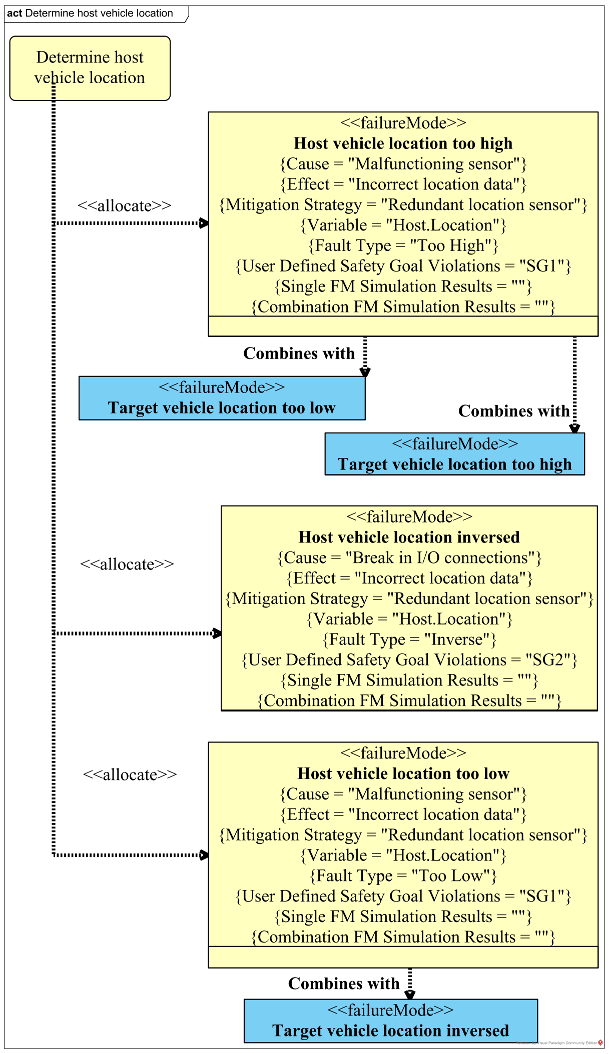

act Determine host vehicle location
Determine host vehicle location
<<allocate>>
<<failureMode>>
Host vehicle location too high
{Cause = "Malfunctioning sensor"}
{Effect = "Incorrect location data"}
{Mitigation Strategy = "Redundant location sensor"}
{Variable = "Host.Location"}
{Fault Type = "Too High"}
{User Defined Safety Goal Violations = "SG1"}
{Single FM Simulation Results = ""}
{Combination FM Simulation Results = ""}
Combines with
<<failureMode>>
Target vehicle location too low
Combines with
<<failureMode>>
Target vehicle location too high
<<allocate>>
<<failureMode>>
Host vehicle location inversed
{Cause = "Break in I/O connections"}
{Effect = "Incorrect location data"}
{Mitigation Strategy = "Redundant location sensor"}
{Variable = "Host.Location"}
{Fault Type = "Inverse"}
{User Defined Safety Goal Violations = "SG2"}
{Single FM Simulation Results = ""}
{Combination FM Simulation Results = ""}
<<allocate>>
<<failureMode>>
Host vehicle location too low
{Cause = "Malfunctioning sensor"}
{Effect = "Incorrect location data"}
{Mitigation Strategy = "Redundant location sensor"}
{Variable = "Host.Location"}
{Fault Type = "Too Low"}
{User Defined Safety Goal Violations = "SG1"}
{Single FM Simulation Results = ""}
{Combination FM Simulation Results = ""}
Combines with
<<failureMode>>
Target vehicle location inversed

FIGURE 9. Failure modes for the "Determine host vehicle location" function before the safety verification is performed. Each failure mode contains a tagged value for storing the results from the fault injection simulation. The syntax for how the results are stored is shown in the safety profile. The failure modes can be combined with the failure modes of other functions, shown in blue, and the relationship is captured as a dependency in the SysML model.

driving environments, scenario generation, and the complete control of actors (pedestrian and vehicles) in the simulator. It is a high-fidelity simulator that can create detailed representations of different environments such as urban layouts, buildings, street signs, highways, etc., for a range of weather and time of day conditions. The simulation platform allows users to set up various sensors and gather data, such as GPS coordinates, speed, acceleration, camera images, radar data, and inertial measurement unit (IMU) data of an automotive system.

CARLA's architecture enables great flexibility and realism for graphical rendering and physics simulation. It is implemented as a layer over the Unreal Engine 4 (UE4) [54], which provides state-of-the-art rendering quality and hyper

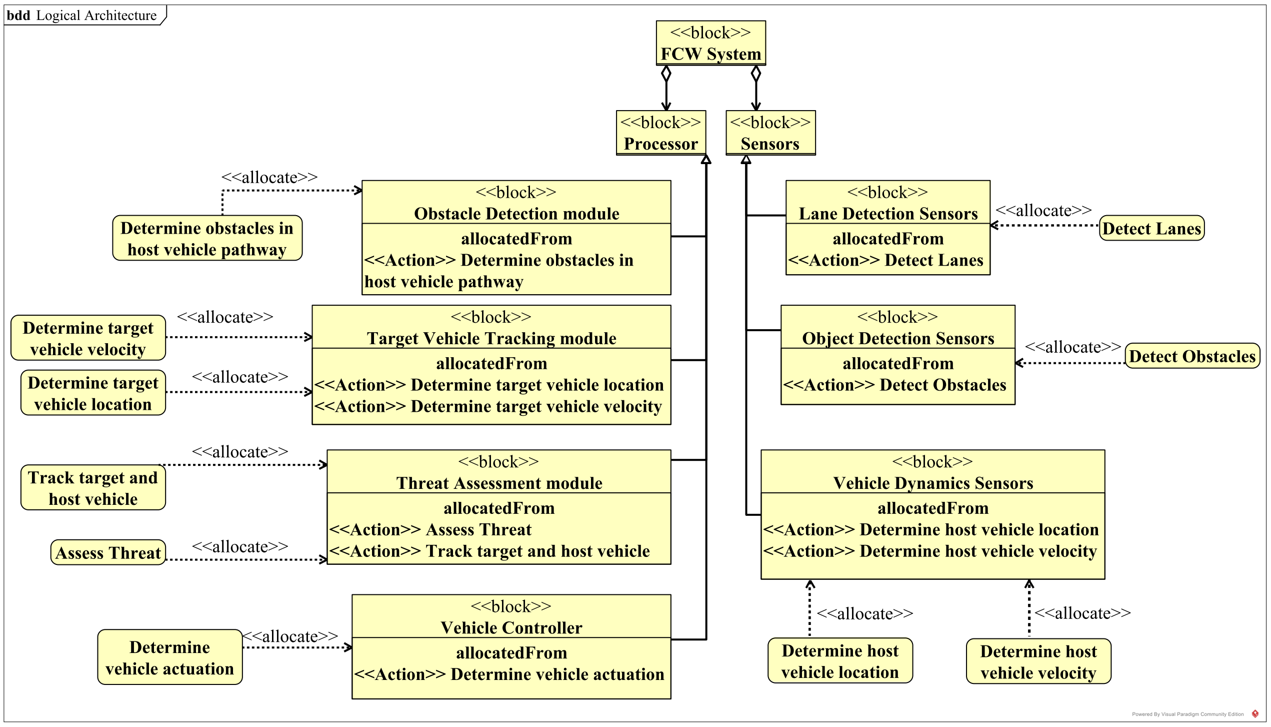

bdd Logical Architecture
<<block>> FCW System
<<block>> Processor
<<block>> Sensors
<<allocate>>
Determine obstacles in host vehicle pathway
<<block>> Obstacle Detection module
allocatedFrom
<<Action>> Determine obstacles in host vehicle pathway
<<block>> Lane Detection Sensors
allocatedFrom
<<Action>> Detect Lanes
Detect Lanes
Determine target vehicle velocity
Determine target vehicle location
<<block>> Target Vehicle Tracking module
allocatedFrom
<<Action>> Determine target vehicle location
<<Action>> Determine target vehicle velocity
<<block>> Object Detection Sensors
allocatedFrom
<<Action>> Detect Obstacles
Detect Obstacles
Track target and host vehicle
Assess Threat
<<block>> Threat Assessment module
allocatedFrom
<<Action>> Assess Threat
<<Action>> Track target and host vehicle
<<block>> Vehicle Dynamics Sensors
allocatedFrom
<<Action>> Determine host vehicle location
<<Action>> Determine host vehicle velocity
Determine vehicle actuation
<<block>> Vehicle Controller
allocatedFrom
<<Action>> Determine vehicle actuation
Determine host vehicle location
Determine host vehicle velocity
Powered By Visual Paradigm Community Edition

FIGURE 10. Logical architecture for the FCW system modelled using a BDD. The BDD shows the different logical blocks of the system and the functions allocated to each block. Each function in the functional architecture is allocated to a block in the logical architecture.

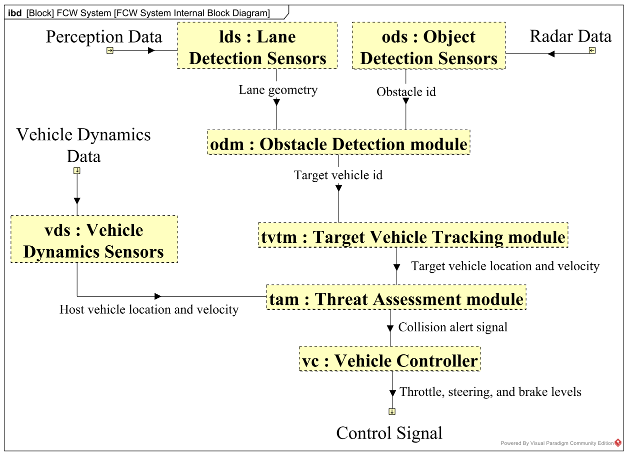

ibd [Block] FCW System [FCW System Internal Block Diagram]
Perception Data
lds : Lane Detection Sensors
ods : Object Detection Sensors
Radar Data
Lane geometry
Obstacle id
Vehicle Dynamics Data
odm : Obstacle Detection module
Target vehicle id
vds : Vehicle Dynamics Sensors
tvtm : Target Vehicle Tracking module
Target vehicle location and velocity
Host vehicle location and velocity
tam : Threat Assessment module
Collision alert signal
vc : Vehicle Controller
Throttle, steering, and brake levels
Control Signal
Powered By Visual Paradigm Community Edition

FIGURE 11. Logical architecture for the FCW system modelled using an IBD.

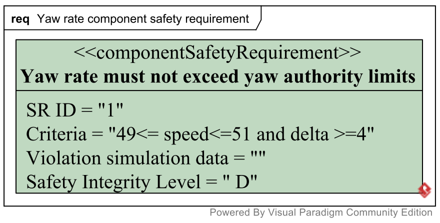

req Yaw rate component safety requirement
<<componentSafetyRequirement>>
Yaw rate must not exceed yaw authority limits
SR ID = "1"
Criteria = "49<= speed<=51 and delta >=4"
Violation simulation data = ""
Safety Integrity Level = " D"
Powered By Visual Paradigm Community Edition

FIGURE 12. Component safety requirement for the vehicle controller block of a forward collision warning system. These low-level requirements are defined to mitigate the functional failure modes. This feature allows traceability from high-level safety requirements (safety goals) to component-level safety requirements.

realistic physics. CARLA can simulate a dynamic world and provide an interface between the world and agent. An agent is usually an autonomous vehicle equipped with a set of sensors to observe its environment and a set of behaviors encoded to achieve a goal. To enable interaction between the world and the agent, the simulator uses a server-client system. The server (also called the world module) runs the simulation and renders the scene. The client module represents the agent developed by the user (i.e., the system under development). The client communicates with the world to obtain information about its surroundings. The client sends commands to the world and obtains sensor readings in return. These commands control the behavior of the automotive system, such as throttle, steering, and brake signals. After executing the command, the world returns the latest sensor readings. Additional commands can be used to control environmental properties such as the weather, illumination, and friction of roadway surfaces. Finally, nonagent automotive systems and pedestrians can be added to the simulation as needed.

The ISDS framework is closely coupled to the CARLA simulator. In its current form, the framework can only assess the safety of systems that can be represented in the simulator, i.e., automotive systems. A CARLA client is developed to represent the system under development using the framework; the client is a skeleton code of the system written in Python. The SysML model is a graphical representation of this client and the system design and safety data from the SysML model are used to configure the parameters in the client. The client is developed using the libraries of the CARLA simulator but contain parameters that are undefined and need to be configured with the parameters defined in the SysML model. It should be noted that this research assumes that the SysML model is an accurate representation of the client in CARLA. Simulating the behavior of an automotive system requires complex Python code, whereas the SysML model is developed at a much higher level of abstraction. It is the responsibility of the engineer to ensure that the functions and logical blocks defined in the SysML model are consistent with those used in the client. Consequently, the framework relies on subject matter experts to validate the models. This does give rise to questions regarding model validation, i.e., "how can one guarantee that the SysML model is consistent with the client?". Several researchers have developed methods for code generation from SysML system models [105, 106]. The SysML models are used to generate code that can be run on specific simulators. However, these approaches are limited to simple systems and require support from the simulators. It is unknown whether such approaches will scale for more complex systems, such as an automotive system within a high-fidelity simulating environment. Consequently, generating code for the entire CARLA client, i.e., code generation from SysML design models, is considered out of scope for this paper but is a potential avenue for future research.

C. FAULT INJECTION ENGINE

The fault injection engine consists of three components: 1) the fault injection configurator (FIC), 2) the fault injection campaign manager (FICM), and 3) the safety metric evaluator (SME).

1) FAULT INJECTION CONFIGURATOR

The FIC is responsible for extracting design and safety data from the SysML model and configuring the CARLA simulator to run the fault injection. The FIC parses the XML file of the SysML model and extracts the data required to set up the configurable parameters in CARLA. Essentially, FIC acts as a bridge between the SysML model and CARLA and ensures that the fault injection simulation reflects the current information in the SysML model. The FIC configures the parameters in the simulator by performing six functions: model selection, configuring failure operational scenarios during which fault injection is performed, configuring safety goal violations, defining component safety requirements, extracting the failure modes to be injected for each block, and translating the failure mode to the fault model. Algorithms 1 (a) and 1 (b) provide the algorithms used in FIC to perform these functions.

1.1 Model Selection

Once the SysML model is populated with the required design and safety data, it is exported as an XML file and parsed by FIC to extract the data required to configure CARLA.

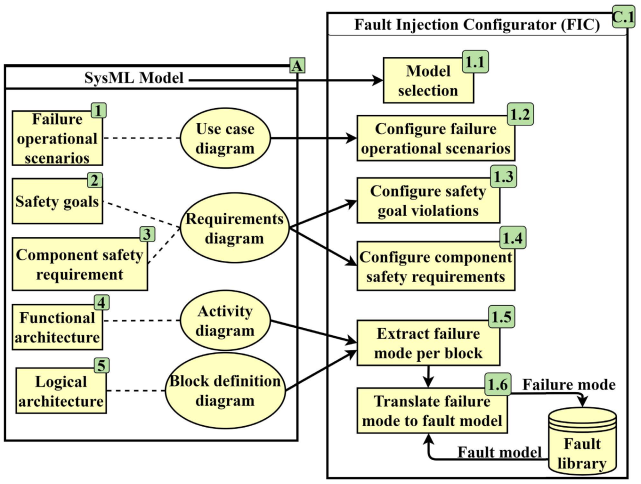

Fault Injection Configurator (FIC)
C.1
SysML Model
A
Failure operational scenarios
1
Safety goals
2
Component safety requirement
3
Functional architecture
4
Logical architecture
5
Use case diagram
Requirements diagram
Activity diagram
Block definition diagram
Model selection
1.1
Configure failure operational scenarios
1.2
Configure safety goal violations
1.3
Configure component safety requirements
1.4
Extract failure mode per block
1.5
Translate failure mode to fault model
1.6
Failure mode
Fault model
Fault library

FIGURE 13. Fault injection configurator (FIC) is responsible for extracting the system design and safety data from the SysML model and configuring the simulator for the fault injection run.

1.2 Configure failure operational scenarios

The use case diagram in the SysML model contains definitions for the configurable parameters of the failure operational scenarios. Fig. 6 illustrates a use case diagram that corresponds to an operational scenario for a forward collision warning system: a host vehicle equipped with a collision alert system encounters a stationery vehicle in its path. To configure this scenario in CARLA, the FIC extracts and initializes the following parameters: the map name (determines the road type), host and target vehicle type, host and target vehicle starting location, host and target vehicle desired speed, and weather conditions. These parameters are defined by an engineer while creating the use case diagram. This function returns a list of objects that represent the different failure operational scenarios of the system.

1.3 Configure safety goal violations

The requirements diagram in the SysML model contains criteria which defines the violation of each safety goal, as illustrated in Fig. 7. The FIC parses the XML file to find the requirements that define the system safety goals. For each safety goal, the identifier, risk, and criteria are extracted to configure the parameters in CARLA. The criteria are converted to a logical expression and evaluated for the Boolean result. If the criteria return true, then the safety goal has been violated. This function returns a list of objects that represent different safety goals of the system.

1.4 Export component safety requirements

The component safety requirements allocated to the blocks in the logical architecture are stored as requirements diagrams (linked as a subdiagram). The FIC iterates through the blocks in the logical architecture and searches for requirements that

Algorithm 1 (a) Algorithm for Section 1.1, 1.2, 1.3 of Fault Injection Configurator (FIC)

```
Input: XML of the SysML model
Output: list of safety goals, component safety
  requirements, and failure operational scenarios.
  SET root = XML Model
  Function configure failure operational scenarios():
    SET list of failure operational scenarios = []
    find all use case elements in the root
    for each use case do
        if stereotype is scenarioConfiguration then
            create the scenarioConfiguration object
            for tagged value in use case do
                store tagged value in object member
                variable
            end for
            append object to list of failure
            operational scenarios
        end if
    end for
    return list of failure operational scenarios
  Function configure safety goals():
    SET list of safety goals = []
    find all requirement elements in the root
    for each requirement do
        if stereotype is Safety Goal then
            create the SafetyGoal object
            for tagged value in requirement do
                store tagged value in object member
                variable
            end for
            append object to list of safety goals
        end if
    end for
    return list of safety goals
  Function configure component safety requirements():
    SET list of component safety requirements = []
    find all requirement elements in the root
    for each requirement do
        if stereotype is a component safety
        requirement then
            create the ComponentSafetyReq object
            for tagged value in requirement do
                store tagged value in object member
                variable
            end for
            append object to list of component safety
            requirements
        end if
    end for
    return list of component safety requirements
```

contain the ≪componentSafetyRequirement≫ stereotype. If defined, the component safety requirement's identifier, allocated block, and criteria are extracted to configure the parameter in CARLA. The criteria are converted to a logical expression and evaluated for the Boolean result during each simulation run. This function returns a list of objects that represent the component safety requirements of the system.

Algorithm 1 (b) Algorithm for Section 1.4 of Fault Injection Configurator (FIC)

```
Input: XML of the SysML model
Output: list of failure modes
  SET root = XML Model
  Function configure failure modes(list of safety goals):
    SET list of failure modes = []
    find all SysMLBLock elements in logical
    architecture
    for each SysMLBLock element do
        Find the allocated function and store address
        for action elements in functional architecture do
            if stereotype is Failure Mode and
                  ↪ address is equal then
                find the allocated failure modes to function
                for each failure mode do
                    create Failure Mode object
                    for tagged value in Failure Mode do
                        store tagged value in object
                    end for
                    append object to list of failure modes
                    find all dependency relationships
                    if dependency is of type Combines with
                          ↪ then
                        Store destination address
                        Find the corresponding failure mode
                        Set failure mode combination
                              ↪ for source and destination
                    end if
                end for
            end if
        end for
    end for
    for each failure mode in list of failure modes do
        find failure mode combinations
        create tuple of failure mode combinations
        append to list of failure modes
        remove duplicate tuples
    end for
    return list of failure modes
```

1.5 Extract failure mode per block

The failure modes defined in the activity diagram during the safety analyses of the system are used to identify the faults that will be injected during the fault injection simulation. The FIC parses the XML file to find the blocks (or subsystems) defined in the logical architecture. The functions allocated to each block are traced to their activity diagram in the functional architecture. The failure mode name, allocated block and function, fault model type, variable to be corrupted, failure mode combinations (if any), and possible safety goal violations (user-defined) are extracted from the XML file. The failure mode combinations are extracted by checking if the failure mode block has a "combines with" dependency relationship with other failure modes. This function returns a list of objects that represent the failure modes and failure mode combinations of the system. Algorithm 1 (b) describes how the FIC module extracts this information from the SysML model.

1.6 Translate failure mode to fault model
The last step is to translate the failure mode (which is described as a string data type) to a fault model that will be injected in CARLA. This translation is done by cross-referencing the failure mode with the list of fault models that have been defined in the fault library. The fault library contains templates for the different fault models that can be injected into the simulator. The fault models define the characteristics of a fault that is to be injected during fault injection. Each fault model contains the following properties: fault type (stuck at, value too high, value too low, delay, inverse value, intermittent loss), percentage change in value due to fault (for too high and too low fault types), number of frames to delay (for stuck at and delay fault types), and the variable in which the fault is injected. The types of faults that are currently supported by the ISDS framework are as follows:

a) Sensor Faults: Sensory faults are injected by manipulating the measurements from sensors (camera, GPS, IMU, etc.). The fault type determines how the measurement is altered (e.g., stuck at faults will maintain the sensor measurement at the constant value that was measured when the fault was triggered). Sensory faults represent scenarios where faulty sensor data are obtained from sensor errors, the environment, noise, etc. Based on the sensor type, these faults alter the host vehicle velocity (faulty IMU sensor), host vehicle location (faulty GPS sensor), and target vehicle velocity and location (faulty radar/camera/lidar sensor).
b) Actuator Faults: Actuator faults are injected by manipulating the output data sent by the controller to the system actuators (throttle, steering, and brake). For example, a "too high" fault type will increase the throttle value by a percentage higher than the correct value. Actuator faults represent real-world scenarios where faulty actuator signals lead to incorrect control commands of a system, such as the unintended acceleration of a vehicle. Based on the actuator type, these faults alter the throttle percentage (faulty accelerator), steering angle (faulty steering system), and brake percentage (faulty brake system).
c) Timing Faults: Timing faults are injected by manipulating the communication of information between systems (sensor to controller or vice versa). Such faults can be injected for sensor, actuator, and processing blocks. The fault types include delayed and loss of data. Timing faults represent real-world scenarios where the communication of data between modules is affected by hardware or software issues.

These fault models provide a basic representation of failure characterization for the supported fault types. The accuracy of the failure characterization can be improved by incorporating concepts such as failure mode uncertainty into the fault model. However, it is beyond the scope of this paper and provides an interesting avenue for future research.

2) FAULT INJECTION CAMPAIGN MANAGER

The FICM is responsible for executing the fault injection. Once the FIC has configured CARLA according to the information from the SysML model, FICM begins executing the fault injection. Each failure operational scenario defined in the use case diagram contains two runs: the golden run (or fault-free run) and the faulty run. The golden run [41] involves observing system behavior without injecting any faults. This data is used to provide a baseline for system behavior in the absence of faults. A faulty run involves injecting the failure mode's fault model and observing the behavior of the system in the presence of faults. The faulty run can involve the injection of single failure modes or failure mode combinations. After each faulty run, the safety metric is computed and stored for the failure mode. Once all the faults have been injected for the current operational scenario, the next operational scenario is loaded, and the process is repeated.

Once the FICM completes the fault injection runs for all the failure operational scenarios defined in the SysML model, the results from the fault injection are updated in the SysML model. This automatic update is enabled by the feedback loops of the framework and is one of the primary contributions of this research. For each failure mode, the safety goal violation, simulation data from the faulty run, and failure operational scenario during which the violation occurred are stored. For faulty runs that involve failure mode combinations, each failure mode stores the names of the combining failure mode, simulation data from the faulty run, safety goal violations, and failure operational scenario during which the violation occurred. Once all the faulty runs are complete, the simulation results for each failure mode are updated under the "Single FM Simulation Results" and "Combination FM Simulation Results" tagged value in the XML file. This XML file can be imported into the SysML tool to be viewed by the engineer. Fig. 14 provides an overview of the entire fault injection process.

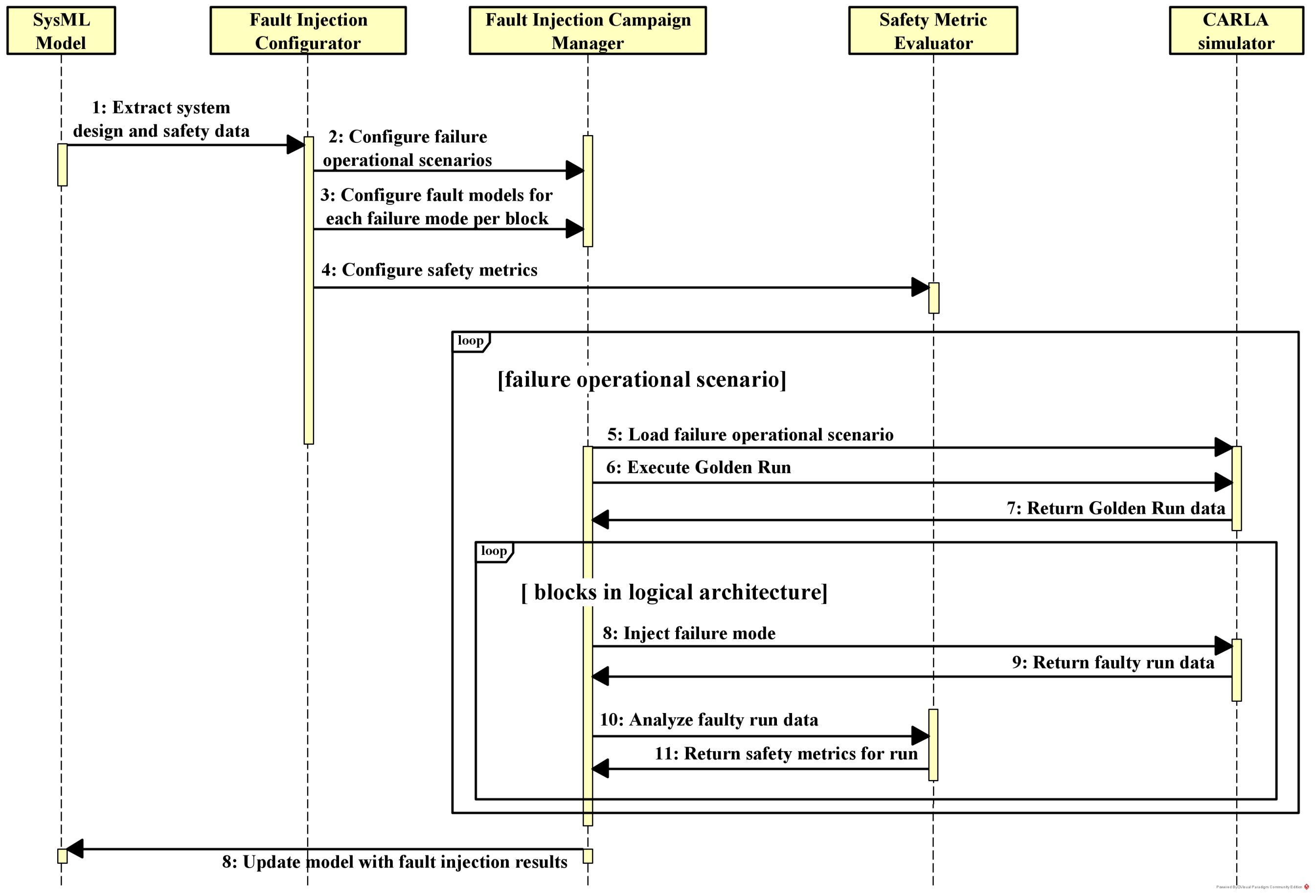

SysML Model
Fault Injection Configurator
Fault Injection Campaign Manager
Safety Metric Evaluator
CARLA simulator
1: Extract system design and safety data
2: Configure failure operational scenarios
3: Configure fault models for each failure mode per block
4: Configure safety metrics
loop
[failure operational scenario]
5: Load failure operational scenario
6: Execute Golden Run
7: Return Golden Run data
loop
[blocks in logical architecture]
8: Inject failure mode
9: Return faulty run data
10: Analyze faulty run data
11: Return safety metrics for run
8: Update model with fault injection results

FIGURE 14. Sequence diagram elaborating the steps involved in performing the safety verification using the fault injection simulation.

3) SAFETY METRIC EVALUATOR

The SME computes the safety metrics configured by FIC. The FICM forward data from the golden and faulty runs to the SME to compute safety metrics. The safety metrics computed by the ISDS framework are as follows:

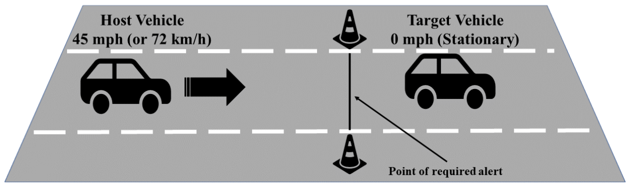
1) Safety goal violations: This is a Boolean value that indicates whether the logical expression that represents the violation of the safety goal, as defined in the SysML model, is met. This returns true if the safety goal was violated. All safety goals defined for the system are evaluated at the end of each faulty run by comparing the golden run data with the faulty run data.
2) Component safety requirement violations: This is a Boolean value that indicates whether the logical expression that represents the violation of the component safety requirement, as defined in the SysML model, is met. This returns true if the component safety requirement was violated. All the component safety requirements are continuously evaluated during each faulty run.

In summary, this section introduced the ISDS framework and provided a detailed description of the model-based safety verification approach used in the framework. The next section describes the application of the ISDS framework to the safety assessment of an FCW system.

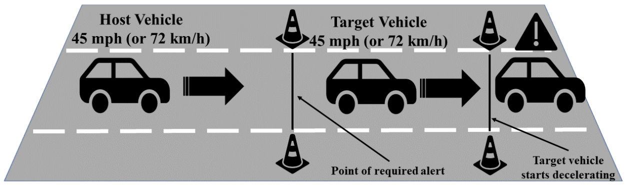
IV. RESULTS: FCW SYSTEM CASE STUDY

This section demonstrates the application of the ISDS framework to the safety assessment of an FCW system. The FCW system case study was selected for several reasons: a) since the ISDS framework is closely coupled to the CARLA simulator, the possible case studies are limited to automotive systems or a safety-critical subsystem within an automotive system; and b) FCW algorithms were first developed in the 1990s [55], [56] and are now ubiquitous in automotive systems; US government agencies are even looking to mandate new vehicles to be equipped with such systems [57]. As a result of the widespread adoption, mature methods have been developed by the National Highway Traffic Safety Administration (NHTSA) to assess the safety of FCW systems. The ISDS framework can be compared with these methods to evaluate the effectiveness of its safety assessment.

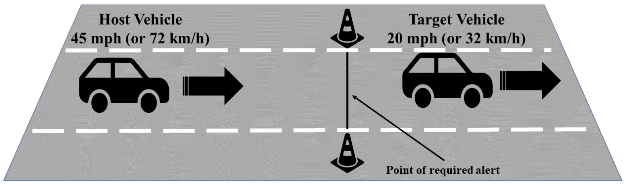
An FCW system identifies the potential for an impending crash situation at the front of a vehicle and either provides an alert to the driver or sends a control signal to activate another vehicular subsystem that can prevent a crash [58].



(a) The host vehicle encounters a stopped target vehicle on a straight road.



(b) The host vehicle encounters a decelerating target vehicle on a straight road.



(c) The host vehicle encounters a slower target vehicle on a straight road.

FIGURE 15. Test scenarios used by NHTSA to evaluate the safety of FCW systems.

Given the widespread adoption of FCW systems, the NHTSA has developed a series of test procedures, called the forward collision warning confirmation test, to evaluate the safety of FCW systems [59]. These tests evaluate the ability of the FCW system to detect and provide an alert for a potential crash under three driving scenarios. Fig. 15 illustrates the scenarios used for the forward collision warning confirmation test. For each scenario, the time-to-collision (TTC) metric is used to assess the risk of collision. TTC is defined as the time required for two vehicles to collide if they continue at their current speed and remain on the same path [60]. The test scenarios developed for the forward collision warning field-tests are as follows:

1) The host vehicle encounters a stopped target vehicle on a straight road, as shown in Fig. 15 (a), where the host vehicle is traveling at 45 mph or 72 km/h and encounters a stationery target vehicle in its path on a straight road. The FCW system of the host vehicle is required to provide an alert when the TTC is $\geq$ 2.1 seconds.
2) The host vehicle encounters a decelerating target vehicle on a straight road, as shown in Fig. 15 (b), where the host vehicle and target vehicle are traveling at 45 mph or 72 km/h, and are 30 meters apart. At a certain point, the target vehicle begins decelerating. For this test, the FCW system of the host vehicle is required to provide an alert when the TTC is $\geq$ 2.4 seconds.
3) The host vehicle encounters a slower target vehicle on a straight road, as shown in Fig. 15 (c), where the host vehicle is traveling at 45 mph or 72 km/h and encounters a slower moving target vehicle in its path, traveling at a constant speed of 20 mph or 32 km/h. The FCW system of the host vehicle is required to provide an alert when the TTC is $\geq$ 2 seconds.

These test procedures have been used by the NHTSA since 2013 to evaluate the safety of FCW systems during field testing. The case study described in this section uses these test procedures to evaluate the safety of an exemplary FCW system design in simulation. The intention of this case study is not to develop a novel FCW system using the ISDS framework but rather to derive the design of an exemplary FCW system from the current literature and use the framework to assess the safety of its design. The architecture, key functions, and subsystems of the exemplary FCW system used in this case study are derived from [61]. The algorithm used by the exemplary FCW system to compute the TTC at which the system warns of an impending collision is based on the Honda algorithm [62], which was chosen for its simplicity, ease of implementation, and extensive use in the field. The remainder of this section is organized as follows: section IV-A describes the results from the application of the left side of the ISDS framework to the FCW system case study, which focuses on system definition and safety analysis using the model-based safety analysis approach. Section IV-B describes the results of the application of the right side of the ISDS framework to the FCW system case study, which focuses on verifying the safety of the FCW system using a model-based safety verification approach.

A. ISDS FRAMEWORK: MODEL-BASED SAFETY ANALYSIS

The application of the framework to this case study begins on the left side of Vee, where the focus is on system definition and safety analysis. As shown in Fig. 1, the model-based safety analysis approach comprises nine steps. After each step, SysML model elements representing system design and safety data are added to the model. Once the SysML model is populated with the data, FMEA tables and fault trees are automatically generated from the model. The remainder of this section will describe the application of each step in the model-based safety analysis approach to the FCW system case study.

1) DEVELOP CONCEPT OF OPERATIONS (CONOPS) AND CREATE FAILURE OPERATIONAL SCENARIOS

This step develops the CONOPS to identify the set of capabilities that the system must have. In this case study, the capabilities are derived from current literature. These capabilities represent high-level needs such as the ability to detect road lanes, detect obstacles in the lane, assess possible threats, etc. The capabilities feed into the system requirements and subsequently the functional architecture. Next, the failure operational scenarios are created using the ≪scenarioConfiguration≫ stereotype to configure the FCW confirmation test scenarios, highlighted in Fig. 15, in the simulator. One of the failure operational scenarios for the FCW system is shown in Fig. 6. This SysML model element captures the information required to recreate the field-test scenarios in the CARLA simulator, such as host and target vehicle speed, location in the map (simulator-specific), weather conditions, etc.

2) IDENTIFY SYSTEM REQUIREMENTS

System requirements translate the system capabilities and stakeholder needs into requirements. These requirements are used to realize the system architecture. However, the system requirements do not directly influence the safety assessment. Since they do not influence the safety assessment, the system requirements diagram is not highlighted in this case study.

3) PERFORM SYSTEM HAZARD AND RISK ASSESSMENT (SYSTEM HARA)

The system HARA consists of three steps. First, system-level hazards that the system might encounter are identified. Next, each hazard is evaluated to identify the risk it poses to the system, and a safety integrity level is assigned to it. Finally, a safety goal is defined to mitigate each hazard. In the FCW system case study, the system-level hazard analysis is performed using the HAZOP technique. Guide words such as late, loss, and not requested are used to identify three system-level hazards: 1) late engagement of the FCW system; 2) unexpected loss of the FCW system; and 3) unexpected engagement of the FCW system. The risk assessment framework from the ISO 26262 safety standard is used to assign a safety integrity level to each hazard. Finally, three safety goals, which represent system-level safety requirements, are defined to mitigate the system-level hazards identified in the previous step. In the SysML model, they are defined using the ≪safetyGoal≫ stereotype. Fig. 7 illustrates the safety goals for the FCW system. The criteria tagged value in the ≪safetyGoal≫ stereotype is used to formalize the requirement and configure the safety goal in the CARLA simulator. For example, the criteria for SG1: Prevent late engagement of FCW system is $golden\_ttc \geq faulty\_ttc \;\; \& \; event == Event.FCW\_ENGAGED$. This expression checks whether the TTC at which the FCW system engaged during the faulty run is less than the TTC at which the FCW system engaged during the golden run.

4) DEVELOP FUNCTIONAL ARCHITECTURE

The functional architecture is developed by translating the system requirements into functions and capturing the interactions between the functions, as shown in Fig. 8. Perception data from camera systems or lidar systems are used to detect lanes by identifying lane markings on roadways. Radar data are used to detect obstacles in front of the vehicle, as well as the obstacle location and velocity. The perception data and radar are combined to determine whether the obstacle, now called the target vehicle, is in the host vehicle's pathway. On-board sensors are used to determine the host vehicle's data, such as velocity and location. The host and target vehicle data are tracked relative to each other and forwarded to assess the threat of collision. Based on the assessment, which will be performed using the Honda algorithm, the vehicle actuation is determined, i.e., the throttle, steering, and brake levels. The levels are translated into control signals and forwarded to the respective actuation subsystems that are external to the FCW system.

5) DEVELOP LOGICAL ARCHITECTURE

The logical architecture identifies the logical blocks of the system (i.e., subsystems), allocates the system functions to the blocks, and identifies the interconnections or data flows between each block. The logical architecture of the FCW system contains seven subsystems with functions from the functional architecture allocated to it, as shown in Fig. 10. The lane detection sensors and obstacle detection sensors are responsible for detecting lane markings on the roadway and obstacles in front of the host vehicle, respectively. The object detection module filters obstacles to those within the lane of the host vehicle. The target vehicle tracking module computes the location and velocity of the target vehicle. The vehicle dynamics sensor computes the location and velocity of the host vehicle. The threat assessment module tracks the relative distance and velocity between the host and the target vehicle. The module also uses the Honda algorithm to determine the threat of a potential collision. Finally, the vehicle controller determines the level of throttle, steering and brake signals required based on the threat assessment and forward the control signals to the external actuation subsystems. Fig. 11 shows the IBD of the logical architecture and illustrates the interconnections and flow of data between subsystems.

6) PERFORM SUBSYSTEM HAZARD AND RISK ASSESSMENT (SUBSYSTEM HARA)

The subsystem hazard analysis is performed on the functions allocated to the different subsystems in the logical architecture using the HAZOP technique. For each function guide words such as too high, too low, lost, delay, intermittent, and inverse are used to brainstorm functional hazards. These hazards represent the function's failure modes and are captured using the ≪failureMode≫ stereotype. Fig. 9 shows the failure modes for the determine host vehicle location function of the vehicle dynamics sensor subsystem. For each failure mode, the cause, effect, user-defined safety goal violations, and mitigation strategy have been defined. Finally, the variable and fault type have been defined to characterize the failure mode in the CARLA simulator. A subset of the failure modes combines with the failure modes of the determine target vehicle location function of the target vehicle tracking module subsystem and are defined using the ''Combines with'' dependency relationship. Similarly, Fig. 16 illustrates the failure modes defined for the determine host vehicle velocity function of the vehicle dynamics sensor subsystem.

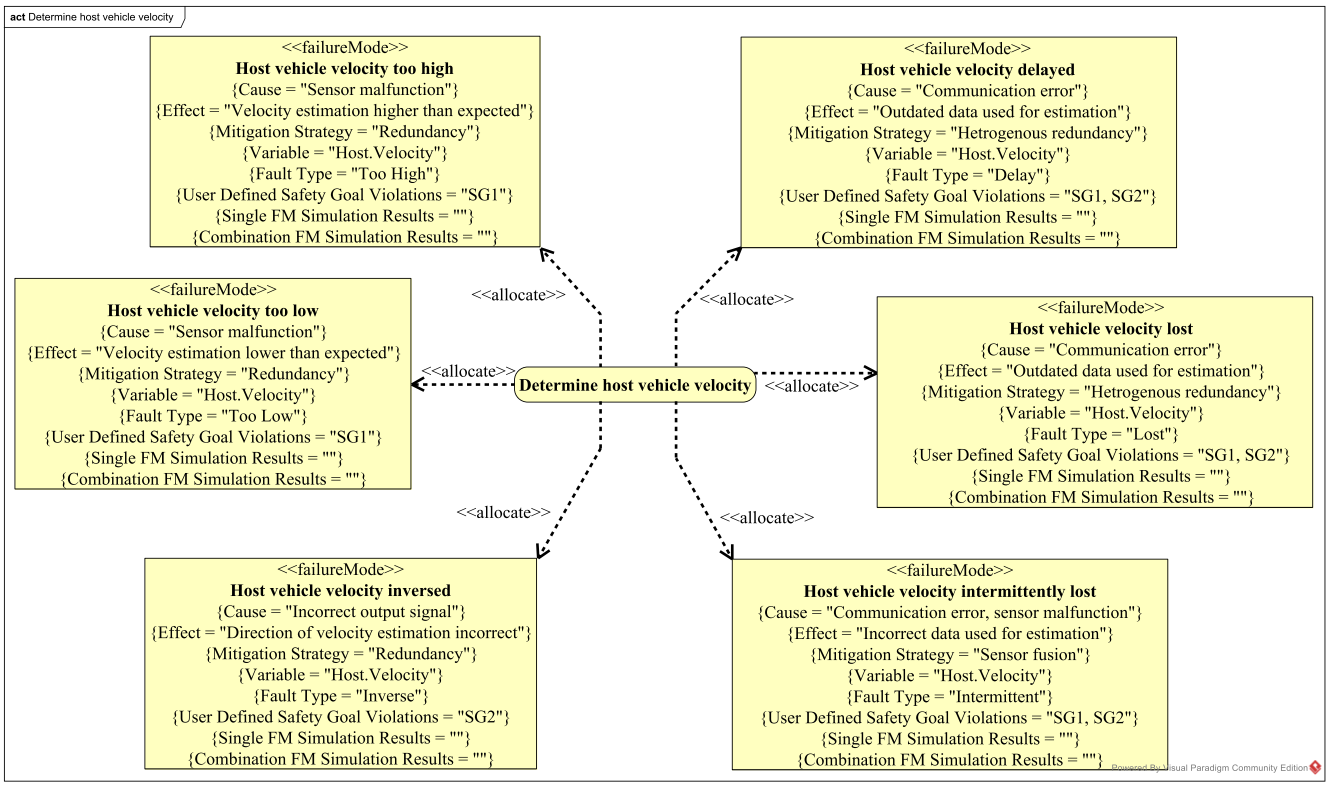

act Determine host vehicle velocity
<<failureMode>>
Host vehicle velocity too high
{Cause = "Sensor malfunction"}
{Effect = "Velocity estimation higher than expected"}
{Mitigation Strategy = "Redundancy"}
{Variable = "Host.Velocity"}
{Fault Type = "Too High"}
{User Defined Safety Goal Violations = "SG1"}
{Single FM Simulation Results = ""}
{Combination FM Simulation Results = ""}
<<failureMode>>
Host vehicle velocity delayed
{Cause = "Communication error"}
{Effect = "Outdated data used for estimation"}
{Mitigation Strategy = "Hetrogenous redundancy"}
{Variable = "Host.Velocity"}
{Fault Type = "Delay"}
{User Defined Safety Goal Violations = "SG1, SG2"}
{Single FM Simulation Results = ""}
{Combination FM Simulation Results = ""}
<<failureMode>>
Host vehicle velocity too low
{Cause = "Sensor malfunction"}
{Effect = "Velocity estimation lower than expected"}
{Mitigation Strategy = "Redundancy"}
{Variable = "Host.Velocity"}
{Fault Type = "Too Low"}
{User Defined Safety Goal Violations = "SG1"}
{Single FM Simulation Results = ""}
{Combination FM Simulation Results = ""}
<<allocate>>
Determine host vehicle velocity
<<failureMode>>
Host vehicle velocity lost
{Cause = "Communication error"}
{Effect = "Outdated data used for estimation"}
{Mitigation Strategy = "Hetrogenous redundancy"}
{Variable = "Host.Velocity"}
{Fault Type = "Lost"}
{User Defined Safety Goal Violations = "SG1, SG2"}
{Single FM Simulation Results = ""}
{Combination FM Simulation Results = ""}
<<failureMode>>
Host vehicle velocity inversed
{Cause = "Incorrect output signal"}
{Effect = "Direction of velocity estimation incorrect"}
{Mitigation Strategy = "Redundancy"}
{Variable = "Host.Velocity"}
{Fault Type = "Inverse"}
{User Defined Safety Goal Violations = "SG2"}
{Single FM Simulation Results = ""}
{Combination FM Simulation Results = ""}
<<failureMode>>
Host vehicle velocity intermittently lost
{Cause = "Communication error, sensor malfunction"}
{Effect = "Incorrect data used for estimation"}
{Mitigation Strategy = "Sensor fusion"}
{Variable = "Host.Velocity"}
{Fault Type = "Intermittent"}
{User Defined Safety Goal Violations = "SG1, SG2"}
{Single FM Simulation Results = ""}
{Combination FM Simulation Results = ""}
Powered By Visual Paradigm Community Edition

FIGURE 16. **Failure modes for the determine host vehicle velocity function of the vehicle dynamics sensor subsystem.**

7) GENERATE SAFETY ARTIFACTS

With the required system design and safety data populated in the SysML model, the FMEA table and fault trees can be automatically generated from the model. The SysML model is exported as an XML file, and the framework's algorithm parses the file to automatically generate safety artifacts. The first safety artifact generated is the FMEA table spreadsheet. The algorithm extracts failure mode data and creates a spreadsheet of the system's FMEA. If the failure mode contains tagged values that have not been defined, the corresponding cell in the spreadsheet contains the "–" symbol, which denotes that an engineer must enter this information in the spreadsheet. Table 3 shows a small section of the FMEA table, which represents the FMEA for the determine host vehicle velocity function.

The next safety artifacts automatically generated by the ISDS framework are the system fault trees, one for each safety goal. Fig. 17 illustrates the fault tree for the violation of the safety goal SG1: Prevent late engagement of the FCW system. Based on the IBD of the FCW system's logical architecture, the fault tree traverses from output to input. Each intermediate level event is either an internal failure of the block or a failure in the input to the block, which is caused by an internal failure of its neighboring connected block. The internal failure of each block is represented using only those failure modes of the block that violate SG1. Since the fault tree for SG1 is large, the base events that contribute to the internal failure of a block are collapsed in Fig. 17. Fig. 18 illustrates the expanded view of the internal failure of the vehicle dynamics block, which contains failure modes for the determine host vehicle velocity function that violate SG1.

At this stage, any changes made to the FMEA by a safety engineer can be automatically updated in the SysML model by using the feedback loop of the framework. This feedback is an important contribution of the model-based safety analysis approach adopted in the ISDS framework and has been highlighted in a previous publication [27]. This feedback prevents any inconsistencies from arising as the design progresses through the system definition phase of the ISDS framework.

8) DERIVE COMPONENT SAFETY REQUIREMENTS

The last step in the model-based safety analysis approach of the framework is to derive the component safety requirements, if necessary. For ease of demonstration, a single component safety requirement is defined in this case study. The determine vehicle actuation function of the vehicle controller block is responsible for computing the level of throttle, brake, and steering signal required based on the input from other subsystems. The failure modes allocated to the function include "Steering signal too high" and "Steering signal too low", "Steering signal lost", "Steering signal delayed", "Steering signal lost intermittently", and "Steering signal inversed". According to [63], a vehicle yaw rate must not exceed 4 degrees per second at a speed close to

TABLE 3. FMEA table for the determine host vehicle velocity function. This table is automatically generated with data from the SysML model, shown in Fig 16. Data in any cell can be updated by a safety engineer, if necessary. Once the results from fault injection are available, the simulation data column is also updated with the results.

Block	Function	Failure Mode	Cause	Effect	Mitigation Strategy	Risk	Safety Goal Violation	Simulation data
Vehicle Dynamics Sensor	Determine host vehicle velocity	Host vehicle velocity too high	Sensor malfunction	Velocity estimation higher than expected	Redundancy	SIL_D	SG1	-
Vehicle Dynamics Sensor	Determine host vehicle velocity	Host vehicle velocity too low	Sensor malfunction	Velocity estimation lower than expected	Redundancy	SIL_D	SG1	-
Vehicle Dynamics Sensor	Determine host vehicle velocity	Host vehicle velocity inversed	Incorrect output signal	Direction of Velocity incorrect	Redundancy	SIL_D	SG2	-
Vehicle Dynamics Sensor	Determine host vehicle velocity	Host vehicle velocity intermittently lost	Communication error, sensor malfunction	Incorrect data used for estimation	Sensor fusion	SIL_D	SG1, SG2	-
Vehicle Dynamics Sensor	Determine host vehicle velocity	Host vehicle velocity lost	Communication error	Outdated data used for estimation	Heterogenous redundancy	SIL_D	SG1, SG2	-
Vehicle Dynamics Sensor	Determine host vehicle velocity	Host vehicle velocity delayed	Communication error	Outdated data used for estimation	Heterogenous redundancy	SIL_D	SG1, SG2	-

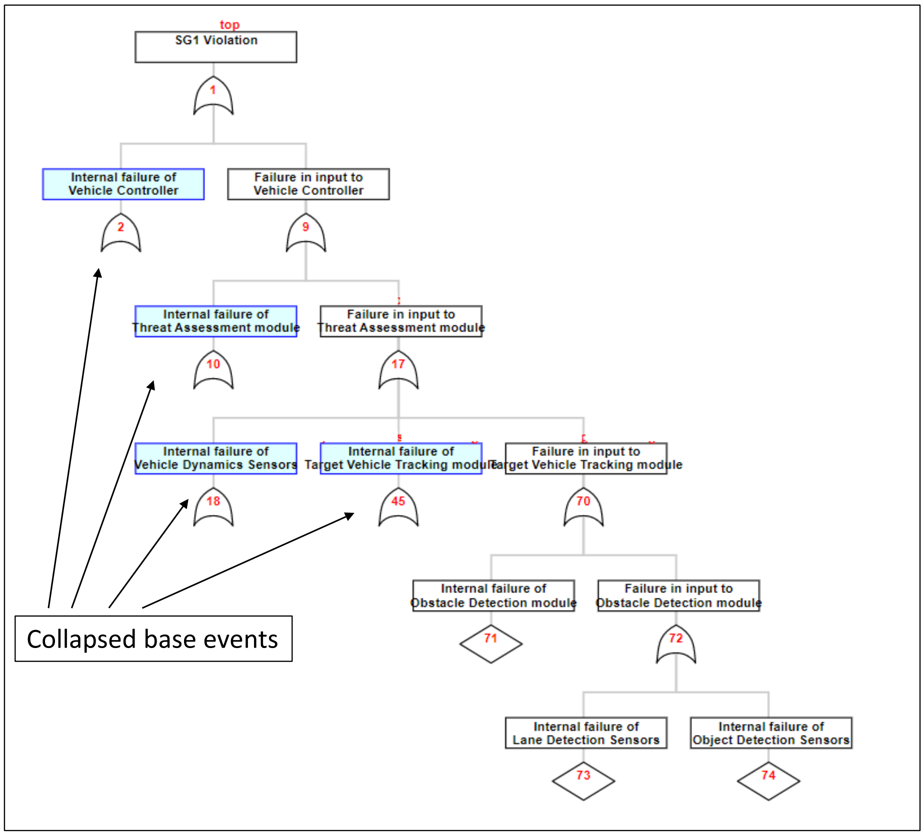

top
SG1 Violation
1
Internal failure of Vehicle Controller
Failure in input to Vehicle Controller
2
9
Internal failure of Threat Assessment module
Failure in input to Threat Assessment module
10
17
Internal failure of Vehicle Dynamics Sensors
Internal failure of Target Vehicle Tracking module
Failure in input to Target Vehicle Tracking module
18
45
70
Internal failure of Obstacle Detection module
Failure in input to Obstacle Detection module
71
72
Internal failure of Lane Detection Sensors
Internal failure of Object Detection Sensors
73
74
Collapsed base events

FIGURE 17. Fault tree for SG1: Prevent delayed engagement of FCW system.

50 km/hr, called the yaw authority limit. To mitigate the effects of the failure modes related to the steering levels of the vehicle controller block, a component safety requirement is defined to prevent the vehicle from exceeding the yaw authority limit. Fig. 12 illustrates the component safety requirement. The component safety requirement is allocated

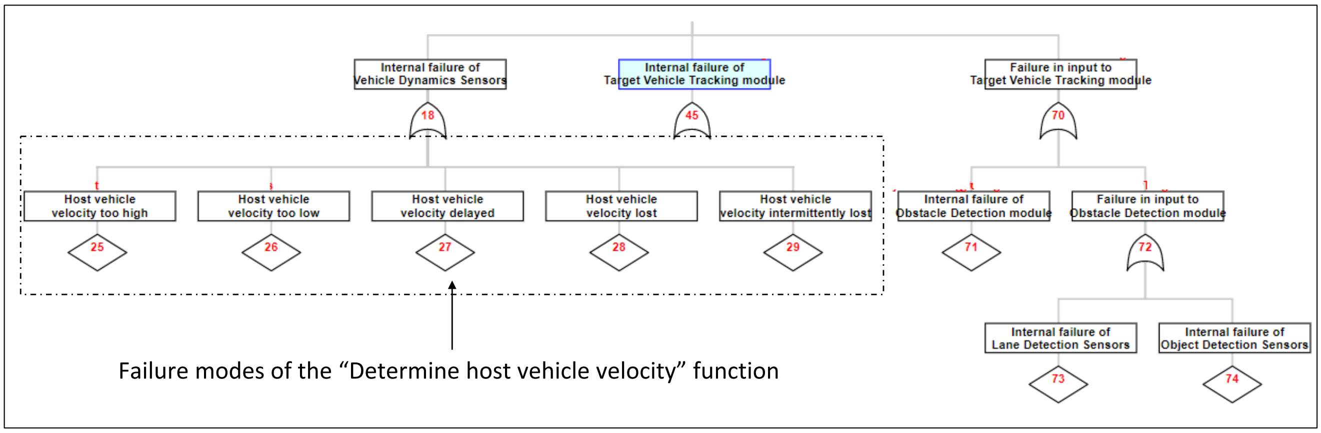

Internal failure of Vehicle Dynamics Sensors
Internal failure of Target Vehicle Tracking module
Failure in input to Target Vehicle Tracking module
18
45
70
Host vehicle velocity too high
Host vehicle velocity too low
Host vehicle velocity delayed
Host vehicle velocity lost
Host vehicle velocity intermittently lost
Internal failure of Obstacle Detection module
Failure in input to Obstacle Detection module
25
26
27
28
29
71
72
Internal failure of Lane Detection Sensors
Internal failure of Object Detection Sensors
73
74
Failure modes of the "Determine host vehicle velocity" function

FIGURE 18. Fault tree for SG1: Prevent delayed engagement of FCW system.

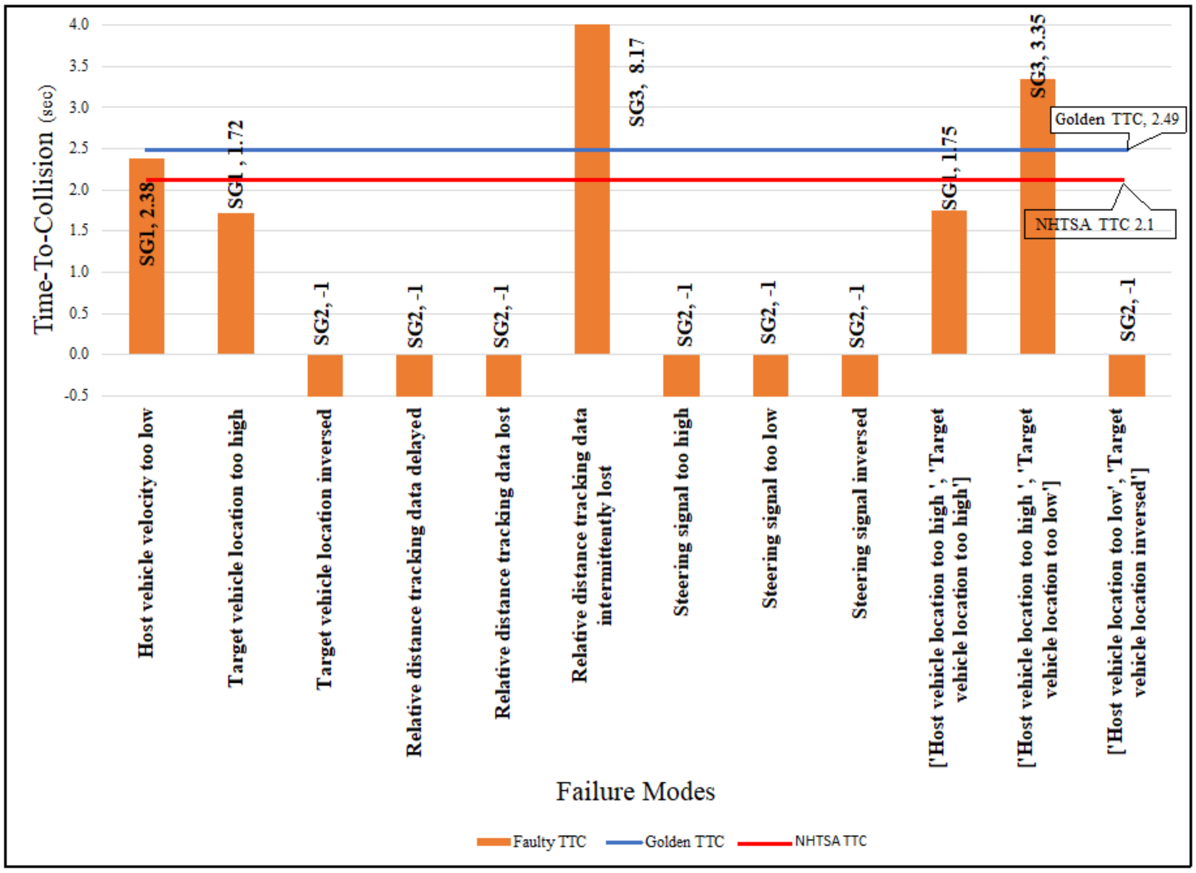


FIGURE 19. Results from safety verification for scenario 1: Host vehicle encounters a stopped target vehicle on a straight road, showing the 12 out of 43 failure mode injections that resulted in a violation of one or more safety goals. The results show the nine behaviors of the FCW system where the faulty TTC, represented by the orange bars, is less than 2.1 seconds and two behaviors of the FCW system where it would pass NHTSA safety assessment but violate the system's own safety goals (i.e., orange bars where the TTC is greater than the golden run TTC value by at least 0.05 seconds).

to the "Steering signal too high" and the "Steering signal too low" failure modes. The component safety requirement's criteria formalizes the requirement in the CARLA simulator.

This step completes the model-based safety analysis approach (i.e., the system definition phase) of the ISDS framework and the SysML model is updated with the required system design data, safety data, and simulation-specific

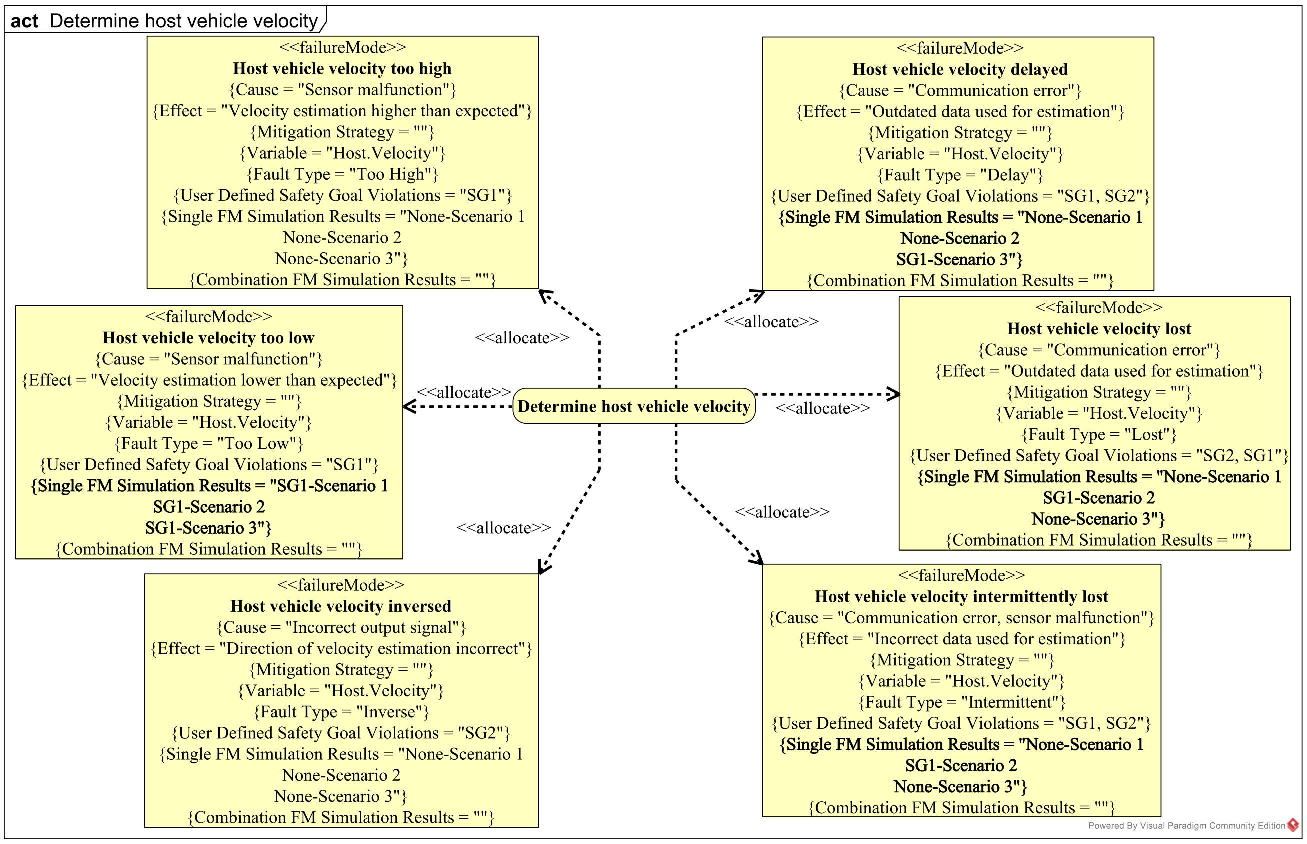

act Determine host vehicle velocity
<<failureMode>>
Host vehicle velocity too high
{Cause = "Sensor malfunction"}
{Effect = "Velocity estimation higher than expected"}
{Mitigation Strategy = ""}
{Variable = "Host.Velocity"}
{Fault Type = "Too High"}
{User Defined Safety Goal Violations = "SG1"}
{Single FM Simulation Results = "None-Scenario 1
None-Scenario 2
None-Scenario 3"}
{Combination FM Simulation Results = ""}
<<failureMode>>
Host vehicle velocity delayed
{Cause = "Communication error"}
{Effect = "Outdated data used for estimation"}
{Mitigation Strategy = ""}
{Variable = "Host.Velocity"}
{Fault Type = "Delay"}
{User Defined Safety Goal Violations = "SG1, SG2"}
{Single FM Simulation Results = "None-Scenario 1
None-Scenario 2
SG1-Scenario 3"}
{Combination FM Simulation Results = ""}
<<failureMode>>
Host vehicle velocity too low
{Cause = "Sensor malfunction"}
{Effect = "Velocity estimation lower than expected"}
{Mitigation Strategy = ""}
{Variable = "Host.Velocity"}
{Fault Type = "Too Low"}
{User Defined Safety Goal Violations = "SG1"}
{Single FM Simulation Results = "SG1-Scenario 1
SG1-Scenario 2
SG1-Scenario 3"}
{Combination FM Simulation Results = ""}
<<allocate>>
<<allocate>>
<<allocate>>
<<allocate>>
<<allocate>>
<<allocate>>
Determine host vehicle velocity
<<failureMode>>
Host vehicle velocity lost
{Cause = "Communication error"}
{Effect = "Outdated data used for estimation"}
{Mitigation Strategy = ""}
{Variable = "Host.Velocity"}
{Fault Type = "Lost"}
{User Defined Safety Goal Violations = "SG2, SG1"}
{Single FM Simulation Results = "None-Scenario 1
SG1-Scenario 2
None-Scenario 3"}
{Combination FM Simulation Results = ""}
<<failureMode>>
Host vehicle velocity inversed
{Cause = "Incorrect output signal"}
{Effect = "Direction of velocity estimation incorrect"}
{Mitigation Strategy = ""}
{Variable = "Host.Velocity"}
{Fault Type = "Inverse"}
{User Defined Safety Goal Violations = "SG2"}
{Single FM Simulation Results = "None-Scenario 1
None-Scenario 2
None-Scenario 3"}
{Combination FM Simulation Results = ""}
<<failureMode>>
Host vehicle velocity intermittently lost
{Cause = "Communication error, sensor malfunction"}
{Effect = "Incorrect data used for estimation"}
{Mitigation Strategy = ""}
{Variable = "Host.Velocity"}
{Fault Type = "Intermittent"}
{User Defined Safety Goal Violations = "SG1, SG2"}
{Single FM Simulation Results = "None-Scenario 1
SG1-Scenario 2
None-Scenario 3"}
{Combination FM Simulation Results = ""}
Powered By Visual Paradigm Community Edition

FIGURE 20. Failure modes of the determine host vehicle velocity function updated with results from safety verification. The "Single FM Simulation Results" tagged values are updated with the safety goal violation and the corresponding failure operational scenario during which the violation occurred.

parameters to perform safety verification. The next section will highlight the results from the model-based safety verification.

B. ISDS FRAMEWORK: MODEL-BASED SAFETY VERIFICATION

This section highlights the results of the model-based safety verification. The ISDS framework uses a simulation-based fault injection technique to verify whether the system design satisfies the safety requirements (safety goals and component safety requirements). A CARLA client is developed to represent the FCW system designed on the left side of the framework (i.e., during system definition). The model-based safety verification approach is highly automated and does not require any manual activities to be performed by an engineer. The automated safety verification using the CARLA simulator is enabled using the fault injection engine, which is one of the key contributions of this research. Once the safety verification is complete, the framework's algorithms automatically update the SysML model with the results. This feedback ensures consistency between the safety verification, SysML model, and safety analyses (i.e., FMEA tables and fault trees), which is another key contribution of this research.

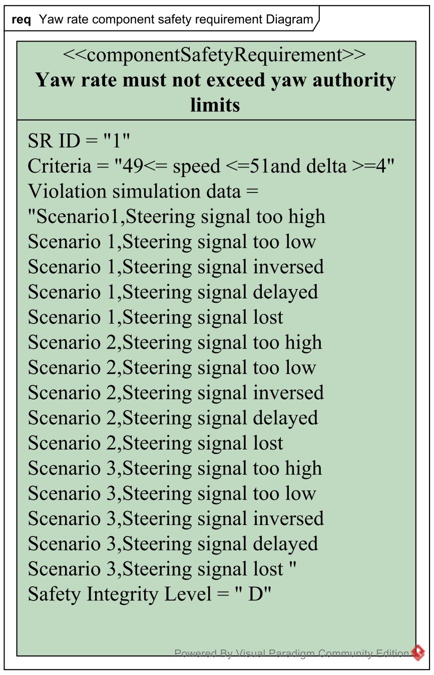
In the FCW system case study, the FIC module of the fault injection engine extracts three failure operational scenario defined using the ≪sceanarioConfiguration≫ stereotype, three safety goals defined using the ≪safetyGoal≫ stereotype, 40 failure modes defined using the ≪failureMode≫ stereotype, three failure mode combinations identified using the "Combines with" dependency, and one component safety requirement defined using the ≪componentSafety Requirement≫ stereotype. For each failure operational scenario, the FICM module of the fault injection engine completes one golden run and 43 faulty runs (40 for single failure modes and three for failure mode combinations). At the end of the golden run, the FICM module computes the TTC at which the FCW system is engaged; this value acts as a reproducible reference or baseline for system behavior in the specific operational scenario. These data represent the FCW system's TTC that would be obtained using the NHTSA forward collision warning confirmation test to assess the safety of the FCW system. For each of the 43 faulty runs, the FICM module injects the failure mode or failure mode combinations and computes the TTC at which the FCW system engages. The SME module of the fault injection engine compares the data from the golden and faulty runs (i.e., the TTC) to evaluate the criteria defined for the three safety goals and one component safety requirement. This analysis determines whether any safety goal or component safety requirements were violated during a faulty run. Once the fault injection is complete for the three failure operational scenarios, the FICM module updates the SysML model with the results.

Fig. 19 illustrates the results of fault injection for failure operational scenario 1. The blue horizontal line represents the TTC for the golden run, called the golden TTC. This indicates the time at which the FCW engaged during the fault-free run. The orange columns represent the TTC for the faulty run, called the faulty TTC. They represent the time at which the FCW system engaged when the corresponding failure mode(s) were injected, shown along the x-axis. The label above each orange column highlights the violated safety goal and the TTC at which the FCW system was engaged. Negative faulty TTC values represent instances when the FCW system failed to engage and did not register a TTC value.

In failure operational scenario 1, the host vehicle equipped with the FCW system encountered a stationary vehicle in its path on a straight road, as illustrated in Fig. 15 (a). The TTC for the golden run, indicated by the horizontal blue line, was 2.49 seconds. The NHTSA assessment requires this value to be ≥ 2.1 seconds. Hence, this system would pass safety assessment based on the NHTSA test criteria. However, using the fault injection engine, of the 43 faulty runs in this scenario (40 faulty runs with single failure mode injection and three faulty runs with dual failure mode injection), 12 runs resulted in a violation of one or more safety goals. Fig. 19 only highlights the data from the 12 runs. The seven negative faulty TTC values, shown in Fig. 19, represent instances when the FCW system failed to engage and did not register a TTC value. During the injection of these seven failure modes, the FCW system encountered the unexpected loss of FCW system hazard and violated SG2: prevent unexpected loss of FCW system.

Since the NHTSA assessment criteria only require the TTC ≥ 2.1 seconds, only nine out of the 12 faulty runs would fail its safety assessment. Since there is no upper limit to the criteria, it is difficult to classify whether runs with a large faulty TTC value would pass or fail safety assessment based on the NHTSA assessment. In Fig. 19, two runs fall into this category, with faulty TTCs of 8.17 seconds and 3.35 seconds respectively. Hence, for scenario 1, in an FCW system that passes the NHTSA safety assessment (based on the golden run data), the ISDS framework can identify at least nine behaviors (characterized by the system's failure modes) where the FCW system would fail the NHTSA safety assessment and two other behaviors where it would pass NHTSA safety assessment but violate the system's safety goals. This demonstrates that the ISDS framework can identify safety-related design issues that impact the system.

Once the fault injection simulation is complete and the results have been compiled by the fault injection engine, the next step is to update the SysML model with the results using the feedback mechanism of the framework. The feedback mechanism from safety verification to the SysML model is implemented by the FICM module in the fault injection engine. The module updates the XML file that was previously used to configure the CARLA simulator with the results from the safety verification. The updated XML file can be imported into the SysML tool to be viewed by the engineer.


req Yaw rate component safety requirement Diagram

<<componentSafetyRequirement>>
Yaw rate must not exceed yaw authority limits

SR ID = "1"
Criteria = "49<= speed <=51and delta >=4"
Violation simulation data =
"Scenario1,Steering signal too high
Scenario 1,Steering signal too low
Scenario 1,Steering signal inversed
Scenario 1,Steering signal delayed
Scenario 1,Steering signal lost
Scenario 2,Steering signal too high
Scenario 2,Steering signal too low
Scenario 2,Steering signal inversed
Scenario 2,Steering signal delayed
Scenario 2,Steering signal lost
Scenario 3,Steering signal too high
Scenario 3,Steering signal too low
Scenario 3,Steering signal inversed
Scenario 3,Steering signal delayed
Scenario 3,Steering signal lost "
Safety Integrity Level = " D"

FIGURE 21. Yaw rate must not exceed yaw authority limits component safety requirement updated with the results from the safety verification. The "Violation Simulation data" tagged value is updated with the failure operational scenario during which the requirement was violated as well as the failure mode that caused the violation.

Two types of SysML model elements are updated with results from the safety verification - the failure modes and the component safety requirements. Fig. 20 illustrates the failure modes of the determine host vehicle velocity function updated with results from the safety verification. For each failure mode, the "Single FM Simulation Results" tagged value is updated with the safety goal violation and failure operational scenario in which the violation occurred. For example, for the "Host vehicle velocity too low" failure mode, the results show that the failure mode only violated SG1 for scenarios 1, 2, and 3. The results in Fig. 20 are consistent with those of the safety verification, as shown in Fig. 19. The second SysML model element that is updated with the results of the safety verification is the component safety requirements. Fig. 21 illustrates that the component safety requirement yaw rate must not exceed the yaw authority limits updated with the results from the safety verification. The requirement's "Violation simulation data" tagged value is updated with the scenario in which the requirement was violated and the failure mode that was injected when the violation occurred. For the FCW system, all three failure operational scenarios require the vehicle to drive on a straight

TABLE 4. FMEA table for the determine host vehicle velocity function. This table is automatically generated with data from the SysML model, shown in Fig. 20.

Block	Function	Failure Mode	Cause	Effect	Mitigation Strategy	Risk	Safety Goal Violation	Simulation data
Vehicle Dynamics Sensor	Determine host vehicle velocity	Host vehicle velocity too high	Sensor malfunction	Velocity estimation higher than expected	Redundancy	SIL_D	SG1	None-Scenario 1 None-Scenario 2 None-Scenario 3
Vehicle Dynamics Sensor	Determine host vehicle velocity	Host vehicle velocity too low	Sensor malfunction	Velocity estimation lower than expected	Redundancy	SIL_D	SG1	SG1-Scenario 1 SG1-Scenario 2 SG1-Scenario 3
Vehicle Dynamics Sensor	Determine host vehicle velocity	Host vehicle velocity inversed	Incorrect output signal	Direction of Velocity incorrect	Redundancy	SIL_D	SG2	None-Scenario 1 None-Scenario 2 None-Scenario 3
Vehicle Dynamics Sensor	Determine host vehicle velocity	Host vehicle velocity intermittently lost	Communication error, sensor malfunction	Incorrect data used for estimation	Sensor fusion	SIL_D	SG1, SG2	None-Scenario 1 SG1-Scenario 2 None-Scenario 3
Vehicle Dynamics Sensor	Determine host vehicle velocity	Host vehicle velocity lost	Communication error	Outdated data used for estimation	Heterogenous redundancy	SIL_D	SG1, SG2	None-Scenario 1 SG1-Scenario 2 None-Scenario 3
Vehicle Dynamics Sensor	Determine host vehicle velocity	Host vehicle velocity delayed	Communication error	Outdated data used for estimation	Heterogenous redundancy	SIL_D	SG1, SG2	None-Scenario 1 None-Scenario 2 SG1-Scenario 3

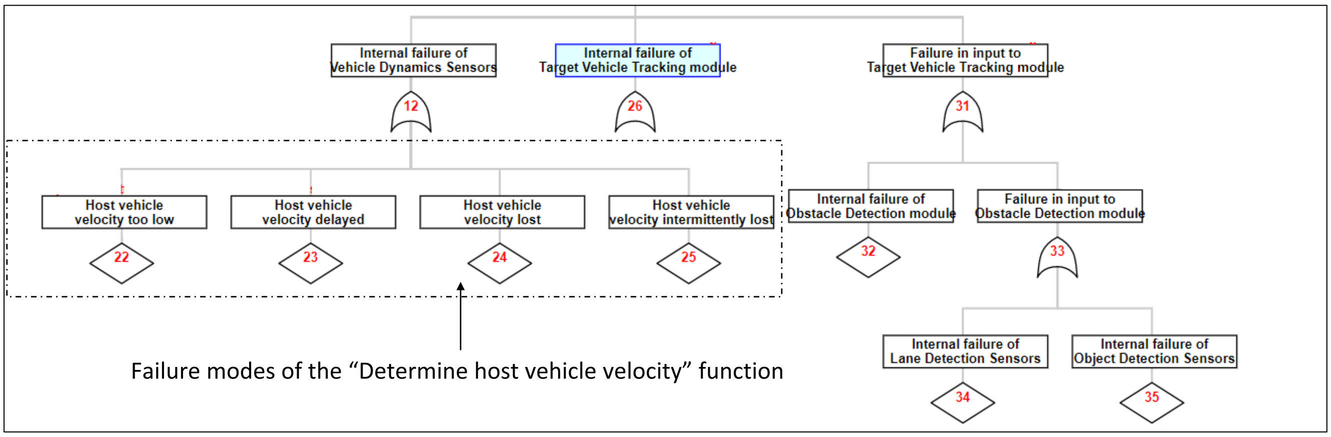

Internal failure of Vehicle Dynamics Sensors
Internal failure of Target Vehicle Tracking module
Failure in input to Target Vehicle Tracking module
12
26
31
Host vehicle velocity too low
Host vehicle velocity delayed
Host vehicle velocity lost
Host vehicle velocity intermittently lost
Internal failure of Obstacle Detection module
Failure in input to Obstacle Detection module
22
23
24
25
32
33
Internal failure of Lane Detection Sensors
Internal failure of Object Detection Sensors
34
35
Failure modes of the "Determine host vehicle velocity" function

FIGURE 22. Fault tree for violation of safety goal SG1 updated with results from safety verification. The fault tree only contains failure modes that were found to violate SG1 during the fault injection simulation.

road. Consequently, the failure modes that affect this component safety requirement are limited to those of the steering actuator, as these are the only failure modes that can cause a sudden change in the yaw rate of the vehicle.

Once the feedback from safety verification to the SysML model is complete, the updated SysML model is used to generate a new FMEA table and three new fault trees of the system. The new FMEA table stores the results from the safety verification in the simulation data column. Not only does this ensure consistency between the SysML model, safety artifact, and safety verification, it also allows the engineer to compare the safety goal violations detected using a formal verification approach against user-defined safety goal violations, which was an analysis based on an engineer's intuition and experience. Table 4 presents a small section of the updated FMEA table, which highlights the FMEA for the "Vehicle Dynamics Sensor" block.

Finally, the fault tree generation algorithm is rerun to generate fault trees based on the safety goal violations found during the fault injection simulation. A fault tree is generated for each safety goal, and each fault tree contains only failure modes that violate the safety goal. The failure modes included in the updated fault tree are based on the updated safety goal violations from safety verification. Fig. 22 illustrates the updated fault tree for the safety goal SG1: Prevent late engagement of the FCW system. The updated fault tree is generated using the results from the safety verification, i.e., using the simulation data. The feedback mechanism of the ISDS framework eliminated the need for engineers to perform this step manually by automating the feedback from safety verification to the SysML model as well as by automatically generating the fault tree. Generating fault trees for each safety goal based on the results from safety verification marks the end of the model-based safety verification approach and completes the model-based safety assessment of the FCW system.

V. CONCLUSION

This paper introduced the ISDS framework, a model-based safety assessment framework, along with a detailed description of the model-based safety verification approach used in the framework. Current safety assessment methods that combine safety analysis and safety verification still require manual tasks to be performed by safety engineers to complete the safety assessment. These tasks include manually generating safety artifacts from a system design model for safety analysis, updating the system design model with changes made to the safety artifact, and updating the system design model with the results from safety verification. These manual tasks are time-consuming and error prone, which can lead to inconsistencies between the system design model, safety analysis, and safety verification.

The main objective of the ISDS framework is to eliminate tasks that can introduce inconsistencies into the safety assessment process. The feedback mechanism of the ISDS framework eliminated the need for engineers to manually update the SysML model with changes made to the safety artifact generated during safety analysis and to manually update the SysML model with results from safety verification. Consequently, the ISDS framework eliminates a key source of inconsistency in the safety assessment process. The automated feedback mechanism combined with the ability to automatically generate FMEA tables and fault trees allows the ISDS framework to capture the iterative nature of safety assessment while maintaining consistency between the SysML model, safety analyses (i.e., safety artifacts), and safety verification. The feedback mechanism of the ISDS framework is a key contribution of this research.

Another key contribution of this paper is the safety verification approach of the ISDS framework. The approach uses a fault injection engine, which allows engineers to inject faults into a system design and observe the system behavior under fault-free and faulty conditions. This allows engineers to assess the safety of the system early in the life cycle using available system design models and to identify safety-related design issues that impact the system. Such assessments can complement existing safety assessment methods that are based on field-testing to provide a comprehensive analysis of the safety of the system.

There are also several avenues for future work that can address the current limitations of this research and extend the ISDS framework. Currently, the ISDS framework is closely coupled to the CARLA simulator and is limited to the evaluation of automotive systems that can be characterized in CARLA. Although the results from the case study are promising, the framework must be evaluated for a broad range of conditions and applications before it can be generalized across other domains or applications. Future research can look at adapting the fault injection engine of the ISDS framework to other high-fidelity simulators across different domains.

Next, the framework relies heavily on subject matter experts for model validation. A promising avenue for future work includes the incorporation of more comprehensive model validation techniques into the framework.

Another possibility is to support the automated generation of executable code from SysML models. The use of automatic code generation will allow engineers to completely generate the CARLA client from the SysML model and remove the responsibility of ensuring consistency between the code running in the simulator and the SysML model from the design and safety engineers. This could also aid in model validation.

Finally, the fault coverage is limited to the failure modes identified in the FMEA and the fault models that are supported by the framework for the CARLA simulator. In addition, the failure modes are deterministic and do not account for any uncertainty. Future work can involve extending the types of faults that can be injected into the CARLA simulator by the fault injection engine. This can help improve the fault coverage of the framework and support an increased number of fault models for automotive systems. Additionally, modeling uncertainty in failure modes will help in injecting failure modes that are more representative of real-world failures. Similarly, modeling uncertainty into failure operational scenarios can help bridge the gap between scenarios executed in simulation and real-world scenario that impact the safety of the system. Once these features are implemented, an interesting avenue for research could be to investigate the statistical characterization of the performance of the framework over a large number of simulations.

REFERENCES

[1] M. Cummings and D. Britton, "Regulating safety-critical autonomous systems: Past, present, and future perspectives," in *Living With Robots*. Amsterdam, The Netherlands: Elsevier, Jan. 2020, pp. 119–140.

[2] Statista. (2019). *Autonomous Cars by Global Market Size 2030 | Statista*. Accessed: Mar. 5, 2021. [Online]. Available: https://www.statista.com/statistics/428692/projected-size-of-global-autonomous-vehicle-market-by-vehicle-type/

[3] Statista. (2020). *Autonomous Guided Vehicle Market Size Worldwide From 2018 to 2026*. Accessed: Mar. 15, 2021. [Online]. Available: https://www.statista.com/statistics/428692/projected-size-of-global-autonomous-vehicle-market-by-vehicle-type/

[4] Z. Tahir and R. Alexander, "Coverage based testing for V&V and safety assurance of self-driving autonomous vehicles: A systematic literature review," in *Proc. IEEE Int. Conf. Artif. Intell. Test. (AITest)*, Oxford, U.K., Aug. 2020, pp. 23–30.

[5] J. Guiochet, M. Machin, and H. Waeselynck, "Safety-critical advanced robots: A survey," *Robot. Auto. Syst.*, vol. 94, pp. 43–52, Aug. 2017, doi: 10.1016/j.robot.2017.04.004.

[6] *Road Vehicles-Functional Safety*, Standard ISO 26262, International Organization for Standardization (ISO), 2018.

[7] C. A. Ericson, *Hazard Analysis Techniques for System Safety*, 2nd ed. Hoboken, NJ, USA: Wiley, 2005.

[8] N. J. Bahr, *System Safety Engineering and Risk Assessment: A Practical Approach*, 2nd ed. Boca Raton, FL, USA: CRC press, 2018.

[9] M. V. Stringfellow, N. G. Leveson, and B. D. Owens, "Safety-driven design for software-intensive aerospace and automotive systems," *Proc. IEEE*, vol. 98, no. 4, pp. 515–525, Apr. 2010.

[10] B. Boehm, R. Valerdi, and E. Honour, "The ROI of systems engineering: Some quantitative results for software-intensive systems," *Syst. Eng.*, vol. 11, no. 3, pp. 221–234, Jun. 2008.

[11] D. K. Hitchins, "Systems engineering: In search of the elusive optimum," *Eng. Manage. J.*, vol. 8, no. 4, pp. 195–207, Aug. 1998.

[12] F. Mhenni, N. Nguyen, and J.-Y. Choley, "Safesyse: A safety analysis integration in systems engineering approach," *IEEE Syst. J.*, vol. 12, no. 1, pp. 161–172, Mar. 2018.

[13] O. Lisagor, T. Kelly, and R. Niu, "Model-based safety assessment: Review of the discipline and its challenges," in *Proc. 9th Int. Conf. Rel., Maintainability Saf. (ICRMS)*, Guiyang, China, Jun. 2011, pp. 625–632.

[14] A. Legendre, A. Lanusse, and A. Rauzy, "Toward model synchronization between safety analysis and system architecture design in industrial contexts," in *Model-Based Safety and Assessment*, vol. 10437. Trento, Italy: Springer, 2017, pp. 35–49.

[15] G. Biggs, T. Juknevicius, A. Armonas, and K. Post, "Integrating safety and reliability analysis into MBSE: Overview of the new proposed OMG standard," in *Proc. INCOSE Int. Symp.*, vol. 28, no. 1, Jul. 2018, pp. 1322–1336.

[16] A. Joshi, S. P. Miller, M. Whalen, and M. P. E. Heimdahl, "A proposal for model-based safety analysis," in *Proc. 24th Digit. Avionics Syst. Conf.*, Oct. 2005, p. 13.

[17] H. Wang, D. Zhong, T. Zhao, and F. Ren, "Integrating model checking with SysML in complex system safety analysis," *IEEE Access*, vol. 7, pp. 16561–16571, 2019.

[18] N. Yakymets, M. Sango, S. Dhouib, and R. Gelin, "Model-based engineering, safety analysis and risk assessment for personal care robots," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Madrid, Spain, Oct. 2018, pp. 6136–6141.

[19] G. Juez Uriagereka, E. Amparan, C. Martinez Martinez, J. Martinez, A. Ibanez, M. Morelli, A. Radermacher, and H. Espinoza, "Design-time safety assessment of robotic systems using fault injection simulation in a model-driven approach," in *Proc. ACM/IEEE 22nd Int. Conf. (MODELS-C)*, Munich, Germany, Sep. 2019, pp. 577–586.

[20] E. R. Carroll and R. J. Malins, "Systematic literature review: How is model-based systems engineering justified?" Sandia Nat. Laboratories, Albuquerque, NM, USA, Tech. Rep. SAND2016-2607627724, 2016.

[21] M. Chodas, "Improving the design process of the REgolith X-ray imaging spectrometer with model based systems engineering," M.S. thesis, Dept. Aeronaut. Astronaut., MIT, Cambridge, MA, USA, 2014.

[22] J. D'Ambrosio and G. Soremekun, "Systems engineering challenges and MBSE opportunities for automotive system design," in *Proc. IEEE SMC*, Banff, AB, Canada, Nov. 2017, pp. 2075–2080.

[23] A. M. Madni and M. Sievers "Model-based systems engineering: Motivation, current status, and research opportunities," *Syst. Eng.*, vol. 21, no. 3, pp. 172–190, May 2018.

[24] A. Joshi and M. P. Heimdahl, "Model-based safety analysis of simulink models using SCADE design verifier," in *Computer Safety, Reliability, and Security* (Lecture Notes in Computer Science), vol. 3688. Berlin, Germany: Springer, 2005, pp. 122–135.

[25] M. Bozzano and A. Villafiorita, "The FSAP/NuSMV-SA safety analysis platform," *Int. J. Softw. Tools Technol. Transf.*, vol. 9, no. 1, pp. 5–24, Feb. 2007.

[26] H. Wang, S. Liu, and C. Gao, "Study on model-based safety verification of automatic train protection system," in *Proc. Asia–Pacific Conf. Comput. Intell. Ind. Appl. (PACIIA)*, Wuhan, China, Nov. 2009, pp. 467–470.

[27] R. Krishnan and S. V. Bhada, "An integrated system design and safety framework for model-based safety analysis," *IEEE Access*, vol. 8, pp. 146483–146497, 2020.

[28] (2020). SEBoK. *System Life Cycle Process Models: Vee—SEBoK*. Accessed: Aug. 4, 2021. [Online]. Available: https://sebokwiki.org/w/index.php?title=System_Life_Cycle_Process_Models:_Vee&oldid=59874

[29] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, "Model checking and the state explosion problem," in *Tools for Practical Software Verification* (Lecture Notes in Computer Science), vol. 7682. Berlin, Germany: Springer, 2011, pp. 1–30.

[30] Ansys. *Ansys Scade Suite: Model-Based Development Environment for Critical Embedded Software*. Accessed: May 8, 2022. [Online]. Available: https://www.ansys.com/products/embedded-software/ansys-scade-suite

[31] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An opensource tool for symbolic model checking," in *Computer Aided Verification* (Lecture Notes in Computer Science), vol. 2404. Berlin, Germany: Springer, 2002, pp. 359–364.

[32] D. Cotroneo, L. De Simone, P. Liguori, and R. Natella, "Fault injection analytics: A novel approach to discover failure modes in cloud-computing systems," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 3, pp. 1476–1491, May 2020.

[33] S. Jha, S. S. Banerjee, J. Cyriac, Z. T. Kalbarczyk, and R. K. Iyer, "AVFI: Fault injection for autonomous vehicles," in *Proc. IEEE/IFIP DSN-W*, Luxembourg, Jun. 2018, pp. 55–56.

[34] S. Jha, T. Tsai, S. Hari, M. Sullivan, Z. Kalbarczyk, S. W. Keckler, and R. K. Iyer, "Kayotee: A fault injection-based system to assess the safety and reliability of autonomous vehicles to faults and errors," 2019, *arXiv:1907.01024*.

[35] G. Li, Y. Li, S. Jha, T. Tsai, M. Sullivan, S. K. S. Hari, Z. Kalbarczyk, and R. Iyer, "AV-FUZZER: Finding safety violations in autonomous driving systems," in *Proc. IEEE 31st Int. Symp. Softw. Rel. Eng. (ISSRE)*, Coimbra, Portugal, Oct. 2020, pp. 1–12.

[36] G. Juez, E. Amparan, R. Lattarulo, A. Ruíz, J. Pérez, and H. Espinoza, "Early safety assessment of automotive systems using sabotage simulation-based fault injection framework," in *Computer Safety, Reliability, and Security* (Lecture Notes in Computer Science), vol. 10488. Trento, Italy: Springer, 2017, pp. 255–269.

[37] P. Koopman and M. Wagner, "Autonomous vehicle safety: An interdisciplinary challenge," *IEEE Intell. Transp. Syst. Mag.*, vol. 9, no. 1, pp. 90–96, Mar. 2017.

[38] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, Apr. 1997.

[39] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection: A survey," *ACM Comput. Surv.*, vol. 48, no. 3, pp. 1–55, Feb. 2016.

[40] M. Kooli and G. Di Natale, "A survey on simulation-based fault injection tools for complex systems," in *Proc. 9th IEEE Int. Conf. Design Technol. Integr. Syst. Nanosc. Era (DTIS)*, Santorini, Greece, May 2014, pp. 1–6.

[41] B. Vedder, "Testing safety-critical systems using fault injection and property-based testing," Ph.D. dissertation, School Inf. Technol., Halmstad Univ., Halmstad, Sweden, 2015.

[42] P. Koopman and M. Wagner, "Challenges in autonomous vehicle testing and validation," *SAE Int. J. Transp. Saf.*, vol. 4, no. 1, pp. 15–24, Apr. 2016.

[43] P. Koopman and M. Wagner, "Toward a framework for highly automated vehicle safety validation," in *Proc. WCX World Congr. Exper.*, Detroit, MI, USA, Apr. 2018, pp.'1–13.

[44] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proc. Conf. Robot Learn.*, Mountain View, CA, USA, 2017, pp. 1–16.

[45] NVIDIA. *NVIDIA DRIVE Sim*. Accessed: Apr. 27, 2021. [Online]. Available: https://www.nvidia.com/en-us/self-driving-cars/simulation/

[46] G. Rong, B. H. Shin, H. Tabatabaee, Q. Lu, S. Lemke, M. Mozeiko, E. Boise, G. Uhm, M. Gerow, S. Mehta, E. Agafonov, T. H. Kim, E. Sterner, K. Ushiroda, M. Reyes, D. Zelenkovsky, and S. Kim, "LGSVL simulator: A high fidelity simulator for autonomous driving," in *Proc. IEEE 23rd Int. Conf. Intell. Transp. Syst. (ITSC)*, Rhodes, Greece, Sep. 2020, pp. 1–6.

[47] A. Pena, I. Iglesias, J. Valera, and A. Martin, "Development and validation of Dynacar RT software, a new integrated solution for design of electric and hybrid vehicles," in *Proc. EVS*, Los Angeles, CA, USA, 2012, pp. 1–7.

[48] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Sendai, Japan, Sep. 2004, pp. 2149–2154.

[49] *Modelica—A Unified Object-Oriented Language for Systems Modeling, Version 3.5*, Modelica Association, 2021.

[50] *SysML-Modelica Transformation, v1.0*, OMG Standard formal/2012-11-09, Object Management Group, Nov. 2012.

[51] A. L. Ramos, J. V. Ferreira, and J. Barcelo, "Model-based systems engineering: An emerging approach for modern systems," *IEEE Trans. Syst., Man, Cybern., C, Appl. Rev.*, vol. 42, no. 1, pp. 101–111, Jan. 2012.

[52] R. Cressent, P. David, V. Idasiak, and F. Kratz, "Increasing reliability of embedded systems in a SysML centered MBSE process: Application to LEA project," in *Proc. M-BED*, Dresden, Germany, Mar. 2010, pp. 1–7.

[53] R. Behjati, T. Yue, S. Nejati, L. Briand, and B. Selic, "Extending SysML with AADL concepts for comprehensive system architecture modeling," in *Modelling Foundations and Applications* (Lecture Notes in Computer Science), vol. 6698. Berlin, Germany: Springer, 2011, pp. 236–252.

[54] *Unreal Engine 4*. Accessed: May 21, 2021. [Online]. Available: https://www.unrealengine.com/en-U.S./

[55] P. B. N. Clarke, "Advanced collision warning systems," in *Proc. IEE Colloquium Ind. Automat. Control, Appl. Automot. Ind.*, London, U.K., Jan. 1998, p. 2.
[56] K. Lee and H. Peng, "Evaluation of automotive forward collision warning and collision avoidance algorithms," *Vehicle Syst. Dyn.*, vol. 43, no. 10, pp. 735–751, Oct. 2005.
[57] P. A. DeFazio. (2020). *H.R.2—116th Congress (2019–2020): Moving Forward Act*. [Online]. Available: https://www.congress.gov/bill/116th-congress/house-bill/2
[58] G. J. Forkenbrock and B. C. O'Harra, "A forward collision warning (fcw) performance evaluation," in *Proc. Int. Tech. Conf. Enhanced Saf. Vehicles (ESV)*, Stuttgart, Germany, 2009, pp. 1–12.
[59] *Forward Collision Warning System Confirmation Test*, N. H. T. S. Admin., Office Vehicle Saf., Office Crash Avoidance Standards, Nat. Highway Traffic Saf. Admin., Washington, DC, USA, 2013.
[60] R. Van Der Horst and J. Hogema, "Time-to-collision and collision avoidance systems," in *Proc. ICTCT Workshop*, Salzburg, Austria, 1993, pp. 109–121.
[61] R. Ervin, J. Sayer, D. LeBlanc, S. Bogard, M. Mefford, M. Hagan, Z. Bareket, and C. Winkler, "Automotive collision avoidance system field operational test report: Methodology and results," NHTSA, Washington DC, USA, Tech. Rep. HS-809 900, 2005.
[62] Y. Fujita, "Radar brake system," *JSAE Rev.*, vol. 16, no. 1, p. 113, Jan. 1995.
[63] A. Neukum, E. Ufer, J. Paulig, and H. Kruger, "Controllability of superposition steering system failures," in *Proc. Steering Tech (TUV SUD)*, Garching, Germany, 2008, pp. 1–12.

RAHUL KRISHNAN (Member, IEEE) received the M.Sc. degree in robotics engineering and the Ph.D. degree in systems engineering from the Worcester Polytechnic Institute (WPI), USA, in 2017 and 2021, respectively. His research interests include model-based systems engineering (MBSE) and functional safety.

SHAMSNAZ VIRANI BHADA (Senior Member, IEEE) received the Ph.D. degree in industrial and systems engineering from The University of Alabama in Huntsville, in 2008. She is currently an Assistant Professor in systems engineering with the ECE Department, Worcester Polytechnic Institute (WPI). Her research interests include applying model-based systems engineering toward safety analysis and policy modeling and digitization.