ORIGINAL ARTICLE

Journal of Software: Evolution and Process

Test Suite Assessment of Safety-Critical Systems using Safety Tactics and Fault-Based Mutation Testing

Havva Gulay Gurbuz[1] | **Bedir Tekinerdogan**[1*] | **Cagatay Catal**[2] | **Nagehan Pala Er**[3]

[1]Information Technology Group, Wageningen University & Research, Wageningen, The Netherlands

[2]Department of Computer Science and Engineering, Qatar University, Doha, Qatar

[3]Aselsan, Ankara, Turkey

Correspondence
Information Technology Group, Wageningen University & Research, Wageningen, The Netherlands
Email: bedir.tekinerdogan@wur.nl

Funding information
No funding received

A safety-critical system is a system in which the software malfunctioning could result in death, injury, or damage to the environment. Addressing safety concerns early on at the architecture design level is critical to guide the subsequent life cycle activities to ensure that the eventual system is reliable. A fundamental approach to address safety at the design level is the adoption of architectural tactics. It is crucial for safety-critical systems to correctly implement the constraints as defined by the selected safety tactics. This article proposes a systematic approach for assessing the adequacy of test suites of safety-critical systems based on these architectural safety tactics. We use a case study to evaluate the effectiveness of our approach using fault-injection techniques. Our study shows that this systematic approach is feasible and effective for test suite assessment of safety-critical systems.

KEYWORDS
software safety, safety-critical systems, fault-based testing, domain-specific language, test suite assessment, safety tactics

1 | INTRODUCTION

Currently, an increasing number of safety-critical systems are controlled by software and rely on the correct operation of the software. Aircraft flight control, nuclear systems, medical devices are well-known examples of safety-critical systems. In this context, a safety-critical system is a system in which software malfunctioning could result in death, injury, or damage to the environment. The software can be considered as safe, which may not lead to a dangerous or life-threatening event for the system. In the literature several studies have discussed the methods, techniques, processes, tools, and models to make the software safe [1, 2, 3].

System safety engineering applies engineering and management principles, criteria, and techniques to optimize all aspects of safety within the constraints of operational effectiveness, time, and cost throughout all phases of the system life cycle [4, 5]. Software safety can be addressed at different levels in the software development life cycle. Addressing safety concerns early on at the software architecture design level is crucial because quality characteristics such as safety cannot be included after the software implementation. An essential approach to address safety at the design level is the adoption of architectural tactics [6]. A tactic is a design decision for realizing quality goals at the architectural level. A safety tactic can be introduced for realizing safety. Wu and Kelly propose, for example, a set of tactics for software safety [7]. Based on the point at which faults are addressed for ensuring safety, we can categorize safety tactics as fault avoidance, fault detection, and fault tolerance safety tactics [8]. Safety-critical systems usually use a combination of these tactics to address the required safety concerns.

Once a safety-critical system is designed, it is crucial to analyze it for safety requirements before starting the implementation, installation, and operation phases. It is critical to ensure that the potential faults can be identified and cost-effective solutions are provided to avoid or recover from the failures. One of the most critical issues is investigating the effectiveness of the applied safety tactics to safety-critical systems. Several scenario-based software architecture analysis approaches [9, 10, 11] exist in the literature to analyze the architecture's quality. Unfortunately, these approaches are general purpose and do not directly consider safety concerns, thus fail to provide an in-depth analysis of the safety tactics.

In this article, we adopt a fault-based testing approach to analyze the effectiveness of the test suite of safety-critical systems using safety tactics. The novelties of this study are pointed out as follows: (1) building a systematic fault-based testing approach for assessing test suite adequacy based on architectural tactics and (2) developing DSL and tool for the proposed fault-based testing approach. An essential aspect in fault-based testing is mutation analysis which involves modifying a program under test to create variants (a.k.a., mutants) of the program. To apply fault-based testing for assessing the test suite, we first present a metamodel and a domain-specific language that models several safety views and the relation to the code. Mutants are generated for the potential hazards and the corresponding tactics. The approach results in the impact analysis of a test suite on the applied tactics. The proposed approach is illustrated using an industrial case study in the avionics domain. The case study demonstrates an important part of the aircraft control platform used in the developed avionics systems. With the case study, our tool allowed us to automate this process by removing manual steps for generating the mutations and running test cases. It also helped us ensure the safety concerns were properly addressed in the test cases by focusing on the safety tactics.

The remainder of the article is organized as follows. In Section 2, we present the required background information for understanding the overall approach. Section 3 provides a case study that we use to illustrate our fault-based testing approach. Section 4 presents the metamodel and domain-specific language for software safety tactics. In Section 5, we present the proposed fault-based testing approach. Section 6 presents our tool that implements the corresponding approach. In Section 7, we illustrate the proposed approach and the tool using the industrial case study. We present DSL evaluation in Section 8. Section 9 presents the related work, and Section 10 concludes the paper.

2 | PRELIMINARIES

2.1 | Safety Tactics

Several studies [7, 12, 13] proposed architectural tactics or patterns for supporting safety design. Safety tactics are organized in [7, 13] based on fault avoidance, fault detection, and fault containment.

Fault avoidance aims to prevent faults from occurring in the system. Simplicity and Substitution are fault avoidance tactics. Fault detection focuses on monitoring the system and identifying faults when they occur in the system. Condition Monitoring, Sanity Check, and Comparison are tactics for fault detection. Fault containment seeks to limit the impact of the fault and prevent propagation of the fault. Fault containment includes Redundancy, Repair, Degradation, Voting, Override, and Barrier tactics. In this study, we refer to and reuse the tactics discussed in the literature [13]. Table 1 shows the safety tactics along with their descriptions.

2.2 | Fault-Based Testing

Fault-based testing is one of the testing approaches which aims to analyze, evaluate, and design test suites by using fault data. Mutation testing is one of the common forms of fault-based testing. It aims to design new test cases by analyzing the quality of the existing test cases. Mutation testing involves modifying a program under test to create variants of the program. Variants are created by making small changes in the program following a pattern. Mutation operators are the patterns to change the program's code, and each variant of the program is called a mutant. A test suite is applied to both a mutant and the original program code. If the original code and mutant behave differently, the test suite can detect the change between the original and the mutant program. However, if the original code and mutant behave the same, the test suite is not adequate to detect the difference, and it needs to be improved.

Mutation analysis consists of the following three steps [14]:

1. Mutation operator selection relevant to faults.
2. Mutant generation.
3. Distinguishing mutants by executing the original program and each generated mutants with the test cases.

After test cases are executed on mutated programs, the mutation score is calculated using the number of live mutants and the number of killed mutants. If the behavior/output of a mutant differs from the original program, the mutant is killed. Otherwise, the mutant is alive. The mutation score is calculated using equation (1). If a mutant's behavior is the same as the original program, the mutant is equivalent. Mutation score [15] is used to evaluate the adequacy of the test cases. The mutation score shows the effectiveness of test cases in terms of their ability to detect injected faults. A higher mutation score means a higher quality of test cases.

$$MutationScore = \frac{\#\,of\ killed\ mutants * 100}{\#\,of\ total\ mutants - \#\,of\ equivalent\ mutants} \tag{1}$$

3 | CASE STUDY

This section describes a case study to illustrate our approach in subsequent sections. The case study is taken from an open-source software called Openpilot [16] implemented using Python and C++. Based on this study [17], it

Safety Tactic	Category	Description
Simplicity	Fault Avoidance	Keep the system as simple as possible to avoid faults.
Substitution	Fault Avoidance	Use more reliable components which are well-proven in safety domain to avoid faults.
Sanity Check	Fault Detection	Check whether a system state or value remains in a valid range defined in system specification.
Condition Monitoring	Fault Detection	Check whether a system value remains in a valid range compared to a more reliable reference value. Reference value is computed at runtime and it is based on system input values and is not pre-known value from the system specification.
Comparison	Fault Detection	Compare the outputs of redundant systems to detect faults.
Diverse Redundancy	Fault Containment	Develop redundant components using different implementations based on the same system specification.
Replication Redundancy	Fault Containment	Develop redundant system using the same implementation.
Repair	Fault Containment	Bring a failed system back to its normal and healthy state and restore it.
Degradation	Fault Containment	Brings a system with an error into a state with reduced functionality in which the system still maintains the core safety functions.
Voting	Fault Containment	Mask the failure through choosing a correct result from redundant systems.
Override	Fault Containment	Choose the output of redundant subsystems by preferring one subsystem or one output state over another.
Barrier	Fault Containment	Protect a subsystem from influences or influencing other subsystems.

TABLE 1 List of Safety Tactics

is one of the most popular open-source software in the safety-critical system domain. Openpilot is open-source driver assistance system developed by Comma.ai. It has Automated Lane Centering, Forward Collision Warning and Lane Departure Warning functionalities supporting a variety of car makes and models. It also has Driver Monitoring capability to alert distracted and asleep drivers. Openpilot consists of different components to communicate with the car and sensors, decide on the state of gas, brake, and steering, and process the sensor data to provide a safer driving experience for the drivers. Their high-level component diagram is given in [18]. In this study, we focus on Driver Monitoring capability which evaluates the data coming from sensors and generates alerts for drivers for a safer driving experience.

In Figure 1, we presented the high-level component diagram of Openpilot that we focused on this study. The overall component diagram for the Openpilot can be found in [18]. *AlertManager* is a module to process and manage the alerts. *Events* is a base module which defines *events* and *alerts* in Openpilot environment. *Controlsd* is a main module to combine wide range of inputs from sensors and car state and produces car-specific Controller Area Network (CAN) messages. CAN is a communication protocol. Electrical units and devices in the car is communicated through CAN messages. *Controlsd* communicates with *AlertManager* to publish proper alerts to the user based on the inputs it receives.

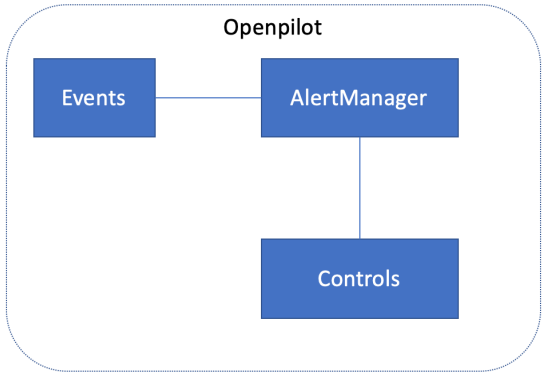

FIGURE 1 High-level architectural diagram of the case study

For the case study, we select *displaying an alert to user when an unusual event occurs*. As explained in [19], to ensure the safety of the driver, hazards and safety requirements should be identified and addressed accordingly. Hazard is a potentially dangerous situation that can result in or contribute to an accident [5]. For the case study, the hazards is *displaying an incorrect alert or no alert to the user*. The possible causes of this hazard are a loss of/error in the car sensors, a loss of/error in communication with the car sensors, an error in the display device, and incorrect evaluation of the data coming from sensors. Car accident is identified as the possible consequence of this hazard. The severity of the hazard is catastrophic since the possible consequence of the hazards is a car crash. Based on this hazard, we define the safety requirements in Table 2.

Openpilot uses several safety tactics in their implementation to meet the defined safety requirements. In order to implemented SR5, they use Sanity Check tactic by checking the event type to decided whether it is an alert or not. If it is not an alert, they do not show it to the user. To implement SR6, they use Condition Monitoring and Sanity Check tactics to monitors the alerts' state and to decide which alert has a high priority to be shown to the user. Table 3 summarizes the rest of the applied tactics for the case study, along with the safety requirements.

Safety Requirement	Explanation
SR1	Events should be evaluated at least from two different components.
SR2	If only one of the components produces an event, the incoming event should be evaluated and a warning should be generated.
SR3	If both components cannot produce an event, the error should be generated.
SR4	The two events should be compared and if they are not same, always the event coming from the selected source should be displayed along with a warning.
SR5	If an event is not an alert, do not show it as an alert.
SR6	If there are multiple alerts that occurring at the same time, show the most recent and high prioritized event as a current alert.
SR7	If an alert is not active, even if it is the most recent and high prioritized one do not show it as a current alert.

TABLE 2 Safety Requirements for the selected hazards

Safety Req.	Safety Tactic & Category	Tactic Description
SR1	Replication Redundancy (Fault Containment)	The events are evaluated by two different Alert Managers where both has the same logic to evaluate events.
SR 2	Condition Monitoring (Fault Detection) Voting (Fault Containment)	The health of both Alert Managers should be periodically monitored to see whether they are healthy or not. If one of them is failing, the event from the other manager will be displayed.
SR3	Condition Monitoring (Fault Detection) Repair (Fault Containment)	The health of both Alert Managers should be periodically monitored to see whether they are healthy or not. If there is a failure on any of the managers, they will be put in repair mode. If both of them are failing, an error will be generated and no alert will be shown.
SR 4	Comparison (Fault Detection) Override (Fault Containment)	If the events produced by each Alert Manager are not same, always show the event coming from Alert Manager 1 and display a warning.
SR5	Sanity Check (Fault Detection)	The given event is checked if it is meets the criteria for representing an alert in the Openpilot environment.
SR6	Sanity Check (Fault Detection) Comparison (Fault Detection)	All the existing alerts' states are monitored and validated against the pre-defined criteria to decide which alert is going to be shown to the user.
SR7	Sanity Check (Fault Containment)	Monitor the alert's state and if it is not active do not show it to the user.

TABLE 3 Applied Safety Tactics to Case Study

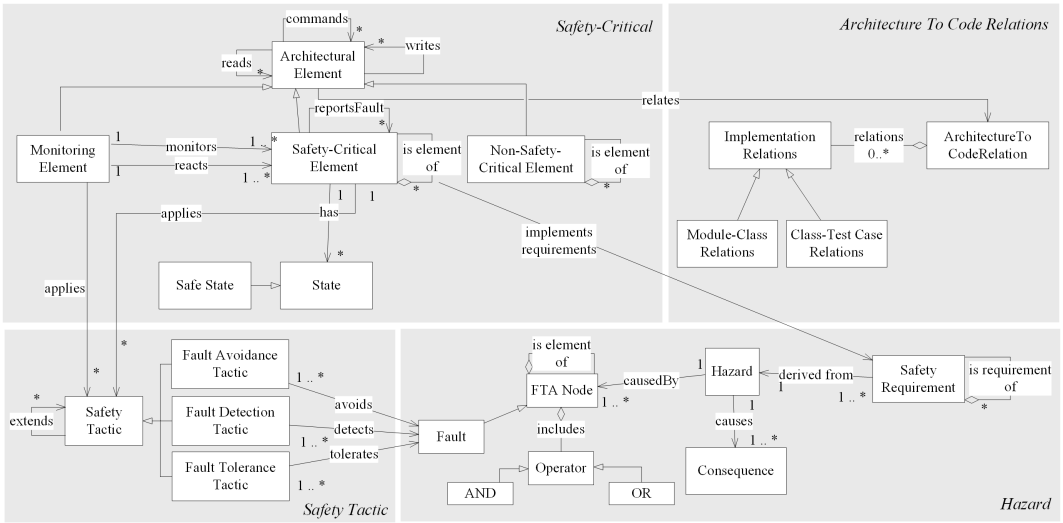

FIGURE 2 Metamodel for Safety DSL

4 | DSL FOR SAFETY

This section presents the metamodel and domain-specific language (DSL) for software safety to represent safety-related concepts. After a thorough domain analysis, our earlier work [8] derived a metamodel to express safety design concepts. In this work, we enhanced the earlier metamodel to support our fault-based testing approach. We updated the previous metamodel by adding *Architecture To Code Relation* part to show concepts and relations used in the fault-based testing approach to generate mutations and running test cases. We present the updated metamodel in Figure 2.

The first part (Safety-Critical) of the metamodel includes the concepts present in the architecture design. Three types of architectural elements are distinguished as *Monitoring Element*, *Safety-Critical Element*, and *Non-Safety Critical Element*. *Monitoring Element* monitors one or more *Safety-Critical Elements* by checking their status. If there is a problem in a *Safety-Critical Element*, the *Monitoring Element* can react by stopping/starting/restarting/initializing the related *Safety-Critical Element*. *Safety-Critical Element* presents the element which includes safety-critical operations. A *Safety-Critical Element* can consist of one or more *Safety-Critical Element*s. We represented this relation in the figure using *is element of*. A *Safety-Critical Element* has *State*s, including *Safe State*. If a fault is detected, which can lead to a hazard in the system and there is a safe state, the system can take itself to the safe state to prevent the hazard. In this regard, we have defined *Safe State* for defining safe states for *Safety-Critical Element*s. A *Monitoring Element* or *Safety-Critical Element* applies *Safety Tactics* in order to ensure the safety of the system.

The second part of the metamodel includes the concepts related to applied safety tactics in the design. We have identified the well-known safety tactics as fault avoidance, fault detection, and fault tolerance. Fault avoidance tactic aims to prevent faults from occurring in the system. When a fault occurs, the fault is detected by applying fault detection tactics. Fault tolerance is the ability of the system to continue correctly and maintain a safe operational condition when a fault occurs. Therefore, applied *Safety Tactic* can be *Fault Avoidance Tactic*, *Fault Detection Tactic*, or *Fault Tolerance Tactic* to deal with faults.

The third part of the metamodel includes the concepts which are related to hazards in the system. A *Hazard*

```
Hazard View Openpilot_HazardView{
    Elements {
        hazard displayingAnIncorrectAlertToUser;

        safetyRequirement doNotDisplayNonAlertEvents;
        safetyRequirement showHighPriAndMostRecentAlert;
        safetyRequirement doNotShowExpiredAlerts;

        consequence carAccident;

    }
    Relations {
        doNotDisplayNonAlertEvents,
        showHighPriAndMostRecentAlert,
        doNotShowExpiredAlerts derivedFrom displayingAnIncorrectAlertToUser;

        displayingAnIncorrectAlertToUser causedBy incorrectAlert;
    }
}
```

FIGURE 3 An example definition of a hazard using our DSL

describes the presence of a potential risk situation that can result or contribute to the mishap. A *Hazard* causes some *Consequences*. *Safety Requirements* are derived from identified *Hazards*. For the safety-critical systems, a thorough hazard analysis should be done to discover potential hazards and identify their root causes. Fault Tree Analysis [1] is one of the most well-known and widely used methods for hazard analysis. It aims to analyze a design for possible faults that could lead to hazards in the system using Boolean logic. We define *FTA Node*, *Operator*, and *Fault* to conducting Fault Tree Analysis. *FTA Nodes*, *Faults*, and *Operators* are the elements of a fault tree. *Operator* is used to conduct Boolean logic. Operator can be AND or OR. One or more *FTA Nodes* cause a *Hazard*.

The last part of the metamodel is *ArchitectureToCodeRelations* defined in the fault-based testing process for mutant generation and test case run steps. As presented in Figure 2, *ArchitectureToCodeRelations* consists of Implementation Relations which can be *Module-Class Relation* or *Class-Test Case Relation*. *Module-Class Relation* describes which *Safety-Critical Elements* defined in *Safety-Critical View* consists of which classes in the program code. *Class-Test Case Relation* defines which classes in the program code should be tested with which test cases. Based on the safety metamodel presented in Figure 2 we provide a domain-specific language (DSL) to represent the concepts in the safety domain. The EBNF grammar [20] of this DSL is presented in A. In Figure 3, we present an example definition of a hazard using our DSL. It illustrates the hazards "displaying an incorrect alert or no alert to the user" and "violating maintaining a safe distance" from the case study that we explained in Section 3.

5 | FAULT-BASED TESTING APPROACH

In the previous sections, we describe the metamodel and the corresponding DSL for modeling safety-critical architectural concerns and their relation to the implementation. Following up on this, Figure 4 shows the process of our fault-based testing approach. Our fault-based testing (FBT) approach leverages the DSL, and we use mutation testing to evaluate the test suite's quality. Within our approach, we enhanced the classical method for mutation testing with
:

- providing a guideline for how to select and decide on mutation operators for applied safety tactics
- automating the mutation generation (for implementation-level mutations), and test case execution
- defining a scope for the testing process by only focusing the safety requirements and the safety tactics

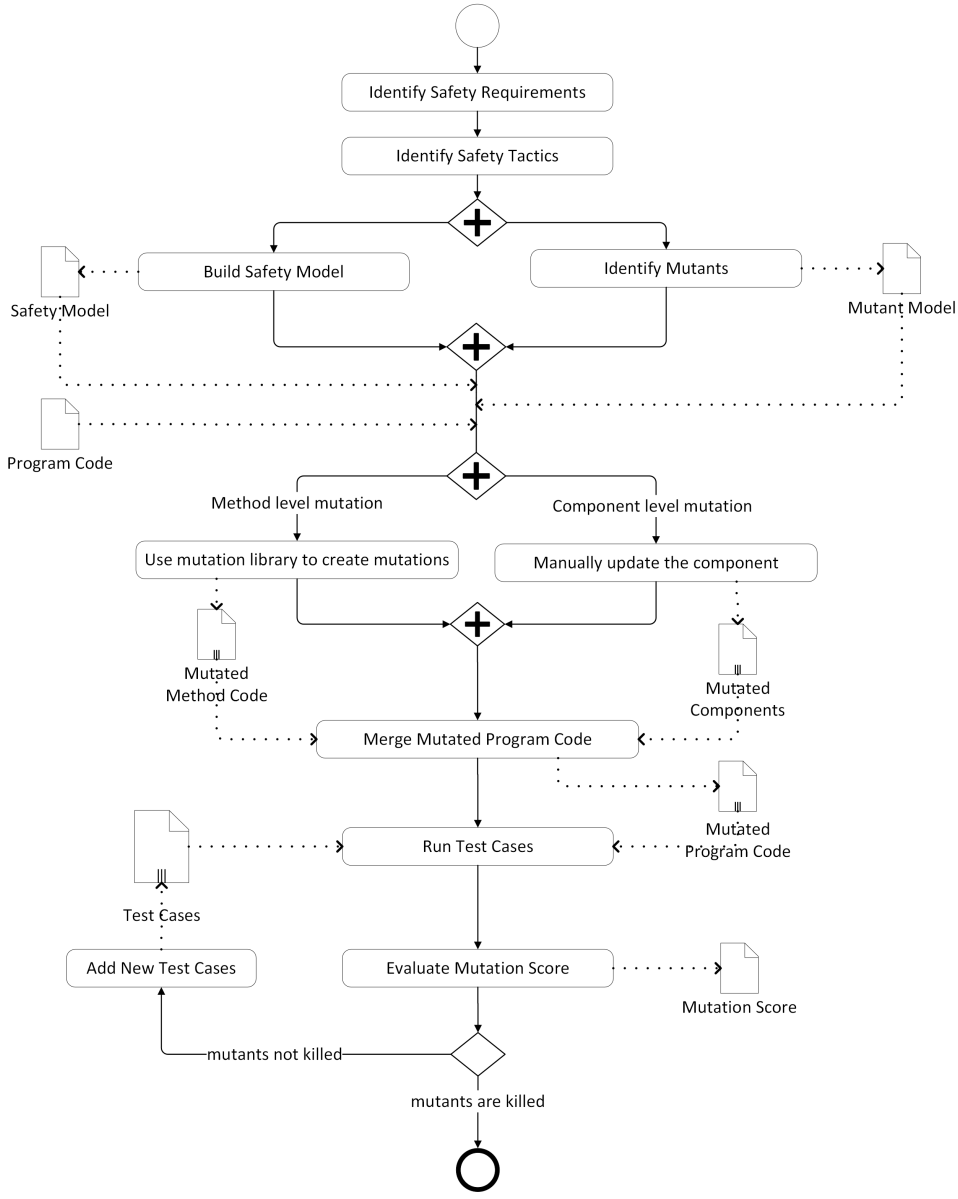

FIGURE 4 Process of Proposed Fault-Based Testing Approach

The approach consists of several steps: identifying safety requirements and safety tactics, building a safety model and mutation model, generating mutants, running the test suite on the generated mutants, and evaluating the results. To build a safety model and mutation model, we first need to define the safety concerns in the system. For this, we start our process by identifying safety requirements and safety tactics. In the following, we explain our proposed approach in detail.

5.1 | Identifying Safety Requirements

The first step of our proposed approach is identifying the safety requirements of the system. Safety requirements are defined based on the hazards and risks in the system [21]. Hazard analysis is performed to identify the hazards in the system by building a list of all hazards, their causes, consequences, and severity. Hazard severity levels are defined as catastrophic, critical, marginal, or negligible in [21]. Hazard identification activity is performed with domain experts (avionics engineers and pilots), system engineers, and safety engineers. The risks in the system are defined by the estimation of the probability of occurrence of each hazard. In [21], occurrence definitions are classified as frequent, probable, occasional, remote, or improbable. Based on the hazard severity and hazard occurrence class identification, risks should be assessed and categorized as high, serious, medium, or low. After the risk definition, a risk assessment should be conducted using fault tree analysis, event tree analysis, simulation, etc. Safety requirements can be derived using identified hazards and risks. In the following subsections, we use "maintaining a safe distance with the leading car" as an example safety requirement. For this requirement, a hazard would be “failing to maintain a safe distance on autopilot mode for autonomous driving cars”.

5.2 | Identifying Safety Tactics

As a second step, safety tactics should be defined to satisfy identified safety requirements. In Table 1, we provided the list of well-known safety tactics that can address safety concerns. Safety tactic(s) should be determined to avoid failures and hazards for each identified safety requirement. Table 1 can be leveraged as a guideline to determine safety tactics for the hazards defined in the identifying safety requirements step. Based on the example we defined Section in 5.1, below are the example safety tactics that can be defined to avoid this hazard using Table 1:

- Sanity Check: Check the distance with the leading car and ensure that it stays within the defined threshold distance.
- Diverse redundancy: Calculate the distance with the leading car from at least two different software components to reduce the risk of miscalculation.

5.3 | Building Safety Model

The next step is creating a safety model using the safety DSL. We use the safety model to generate mutants and run test cases. Hazard view, safety tactic view, and safety-critical view should be defined in order to construct a safety model.

Hazard View

The hazard view should include the safety requirements and the hazards derived from the safety requirements in the first step of our FBT approach. In addition to hazards, the model should contain failures and faults that the identified

hazards can cause. Figure 5 shows a simple hazard view for the example hazard we have defined in the previous subsections.

Safety Tactic View

The safety tactic view consists of the safety tactics identified in the second step of our FBT approach. The safety tactic view should contain the information on "Sanity Check" and "Diverse Redundancy" tactics we defined in Section 5.2. Figure 6 shows a simple safety tactic view for the example scenario we have defined.

Safety-Critical View

The safety-critical view describes the architectural components of the system from a safety perspective. Figure 7 shows a simple safety-critical view for the example scenario we have defined. In this example, since in the system we apply diverse redundancy tactic, we have two distinct components to calculate the distance between the leading car. This view also includes other safety-critical, non-safety-critical and other elements in the system.

Implementation Relations View

For the mutant generation and test case execution steps, architecture to code relations should also be defined. Figure 8 shows a simple architecture to code relationships view for the example scenario we have defined. In this example, with module-class relations, we indicate that the "distanceCalculatorComponent1" includes "Distance" and "CalculatorComponentA" implementation classes/files. And with class-test case relations, we indicate that the tests for "Distance" class/file lives in test suite "distanceTests" where the tests for "CalculatorComponentA" lives in test suites "calculatorTests" and "componentATests".

```
Hazard View HazardViewExample{
    Elements {
        hazard failingToMaintainSafeDistance;

        safetyRequirement maintainSafeDistance;

        consequence carAccident;

        fault lossOfCarSensor;
        fault errorInCarSensor;
        // more faults ...

        faultTree incorrectDistanceCalculation(
         // fault tree definition

        );
    }

    Relations {
        maintainSafeDistance derivedFrom failingToMaintainSafeDistance;

        failingToMaintainSafeDistance causedBy incorrectDistanceCalculation;

    }
}
```

FIGURE 5 Hazard View for a sample hazard

```
SafetyTacticView SafetyTacticsViewExample{
    faultContainment redundantDistanceCalculator {
        type="DiverseRedundancy"
        containedFaults= // faults

    };


    faultDetection safeDistanceCheck {
            type="SanityCheck"
            detectedFaults= // faults

    };
}
```

FIGURE 6 Safety Tactic View for the example scenario

```
Safety-CriticalView SafetyCriticalViewExample{
    Elements {

        safety-critical distanceCalculatorComponent1{
            criticalityLevel=B;
            implementedSafetyRequirements= maintainSafeDistance;
            implementedTactics= redundantDistanceCalculator;
        };

        safety-critical distanceCalculatorComponent2{
            criticalityLevel=B;
            implementedSafetyRequirements= maintainSafeDistance;
            implementedTactics= redundantDistanceCalculator;
        };

        // other elements
    }
    Relations {
        // elements relations

    }
}
```

FIGURE 7 Safety-Critical View for the example scenario

```
ImplementationRelations {
    Module-Class Relations{
        distanceCalculatorComponent1 composesOf = {Distance, CalculatorComponentA }
        distanceCalculatorComponent2 composesOf = {Distance, CalculatorComponentB }
    };


    Class-Test Case Relations{
        Distance testWith = {distanceTests};
        CalculatorComponentA testWith = {calculatorTests, componentATests};
        CalculatorComponentB testWith = {calculatorTests, componentBTests};
    };
}
```

FIGURE 8 Architecture to Code Relationships View for the example scenario

Safety Tactic	Tactic Properties	Mutation Model
Simplicity	N/A	N/A
Substitution	- Using more reliable components	- Architectural-level mutation: Replace the components with faulty components
Sanity Check	- Checking the system state or value to see if it remains within a valid range determined in the system specification	- Implementation-level mutation: Mutate the implementation of checking logic by replacing, adding or removing the arithmetic, relational and logical operators Mutation operators: Arithmetic operators, Logical operators, Relational operators
Condition Monitoring	- Checking the system state or value to see if it remains within a valid range calculated at run time - Having a component for monitoring	- Mutate the implementation of checking logic by replacing, adding or removing the arithmetic, relational and logical operators Mutation operators: Arithmetic operators, Logical operators, Relational operators - Architectural-level mutation: Replace the monitoring component with the faulty one
Comparison	- Comparing values from redundant ones	- Implementation-level mutation: Mutate the implementation of comparison logic by replacing, adding, removing the relational and logical operators Mutation operators: Logical operators, Relational operators
Diverse Redundancy	- Having different implementations for redundant components or subsystems	- Architectural-level mutation: Replace the redundant component(s) with a component which has a same implementation of the primary component.
Replication Redundancy	- Having same implementation for redundant components or subsystems	- Architectural-level mutation: Replace the redundant component(s) with a component which has different implementation from the primary component.
Repair	- Having a component for repairing the failed components	- Architectural-level mutation: Replace the implementation of a repairing component with a component that has missing functionalities or a faulty one
Degradation	- Having a component that brings the system to a state with reduced functionalities	- Architectural-level mutation: Replace the implementation of a degradation component with a component that has missing functionalities or the faulty one
Voting	- Having a component that chooses the majority of the output values as output	- Implementation-level mutation: Replace the implementation of the voting component to select output value randomly. There is no mutation operator suggested, this requires domain knowledge to mutate the program code.
Override	- Having a component that chooses the output of redundant components by preferring one output over another	- Implementation-level mutation: Replace the overriding logic by selecting the output value from one of the components other than the preferred one.
Barrier	- A barrier component for protecting a component from influences or influencing others	- Architectural-level mutation: Replace the barrier component with one that allows the components to have an effect on each other.

TABLE 4 Mutation Model for Safety Tactics

5.4 | Mutant Generation

In order to generate mutants, we need to know what part of the system needs to be changed and how its behavior is going to be changed. With safety models, we already define what part of the system needs to be changed by focusing on safety tactics and the safety-critical components in the system. We introduce a mutation model for each safety tactic to define the behavioral change. Each mutation model describes the possible ways of changing the behavior of the applied safety tactic. Using these models, the mutation generation can be achieved. Each row in Table 4, explains a mutation model for well-known safety tactics listed in Table 1. Each mutation model is defined based on the tactic properties column. If the safety tactic is addressed on the implementation (code) level, the mutation is also applied on implementation-level. Where if the safety tactic is addressed on the component level, the mutation is applied at architectural-level. Table 4 also includes common mutation operators related to each implementation-level mutation model. If we take the example safety tactics we defined in Section 5.2, Sanity Check tactic requires having a range check on a system state or value to check their validity. The mutation model for this tactic would be on the implementation-level where we mutate the tactic implementation by adding or removing the arithmetic, relational and conditional operators. If we consider Diverse Redundancy tactic, it requires having redundant components which they have different implementations. The mutation model for this tactic would be on the architectural-level where we add redundant components to have each component have the same implementation. Table 4 can be used as a guideline to build mutation models for each safety tactic defined for the given safety-critical system. The relations between safety tactics, mutation models and mutation operators in Table 4 are addressed in our tool which is presented in Section 6.

There are several tools in the literature to generate implementation-level mutants. *MutPy* is [22] is one of the mutation testing tool for Python 3.3+ for generating implementation-level mutants automatically. We use *MutPy*'s guideline while selecting the proper mutation operators for the safety tactics that we applied to our case study. We have a code generation process where it provides an automated way for creating implementation-level mutations. The code generation process uses the mapping between safety tactic and MutPy mutation operator presented in Table 4. For the code generation process, we have leveraged the code generator provided by Xtext framework [23]. Xtend [24] is part of the Xtext framework, and it is used for model-to-model or model-to-text transformation. We used Xtend to generate a code from our safety model (model-to-text transformation). The code generator is part of the tool we developed which is presented in Section 6. For the code generation process, we need the parts of the program code are going to be mutated and what type of mutation is going to be applied. The code generation process extract this information from the safety model of the system. Below are the steps for generating implementation-level mutations within our tool.

1. Find applied safety tactic(s) for each safety requirement using the hazard and safety tactic view. Create a mapping between safety requirement and associated safety tactics.
2. Find safety-critical modules for each safety tactic using the safety-critical view and create a mapping between them.
3. Find implementation files/classes for each safety-critical modules from implementation relations view and create mapping between them.
4. Find test suites for each implementation file/class from implementation relations view and create mapping between them.

Pseudo code for mutant generation:

for each safety requirement in safety model **do** *// use information from step #1*

```
    for each safety tactic applied for the safety requirement  do // use information from step #1
        for each safety-critical module implements the safety tactic  do // use information from step #2
            code snippet += generate Python code with MutPy library // use information from step #3 and step #4
        end for
    end for
    generate Python file with the code snippet
end for
```

The code generation process uses the mapping between safety tactics and mutation models defined to decide on the mutation behavior. The mutation model tells us what type of mutations are going to be applied. If the mutation is at architectural-level, the mutation generation needs to be performed manually. On the other hand, for the implementation-level mutations, our tool generates a code snippet with the mutation operators defined in Table 4 so that mutants can be generated. In order to get the parts of code to be mutated for implementation-level mutations, our tool extracts the module-class relations and test classes-class relations from Architecture To Code Relations for the safety-critical elements we have obtained. These relations help us to get the implementation details such as module, the class, and the test class. This information indicates the parts of the code are going to be mutated. For each safety tactic in the system, Python code is generated using the extracted information. The generated code is skeleton code which has the required code to generate mutants and run test cases by executing related methods from *MutPy*. We provide the mutation operators to the skeleton code and run the complete code with the original program code. Each selected mutation operator is switched with the operator in the original code by *MutPy*, and mutants are generated. Architectural-level mutation requires adding, removing, or modifying a component. Since it requires implementation-specific knowledge, it can be achieved by updating the component manually, or this process can be fully or partially automated depending on the case.

Considering the example scenario we provided in previous sections, the mutation model for the SanityCheck tactic is on implementation-level and it requires to replace, add or remove the arithmetic, relational and logical operators in the program code. In this case the the generated code snippet for this tactic is given below:

```
mut.py -t calculatorComponentA.py -u calculatorTests.py componentATests.py
       -o AOD AOR COD COI ROR LOR LOD --report-html mutationReport
```

5.5 | Running Test Cases on the Mutants and Mutation Score Evaluation

When we have the mutants generated, as a next step, the test cases are run on the mutants to assess the quality of the test suite. Our study focuses on evaluating the quality of the existing test suite. From this perspective, we use the test suite implemented during the system development. Our approach does not include a process for generating test cases. For implementation-level mutations, this step is also automated. Test cases are run by executing the generated code. For the architectural-level mutations, we run the test cases manually. Test suite evaluation is performed on the implementation level.

Based on the results of the test case execution step, we calculate the mutation score and evaluate it. If there is an alive mutant (not killed by any test cases), we add new test cases to handle the alive mutants. This process is repeated until all the mutants are killed.

6 | TOOL

In this section, we present the tool that we developed in the Eclipse environment to define safety models using safety DSL and the Python script to apply a fault-based testing process.

We defined the grammar of safety DSL using Xtext [23], a language development framework provided as an Eclipse plug-in. After defining our DSL in Xtext, we wrote our code generator using Xtend provided in Xtext framework for the safety DSL. Xtext and the corresponding code generator create the parser and runnable language artifacts. From these artifacts, Xtext generates a full-featured Eclipse text editor. Figure 9 shows the snapshot of the Eclipse text editor for our case study. As explained in the previous section, for the mutant generation and test case execution steps, an existing open-source Python project *MutPy* is used. *MutPy* provides mutation operators for the mutant generation. Additionally, it enables to execution of predefined test cases on mutated program code. The Python script is generated during the code generation process to mutate the program code and execute test cases leveraging *MutPy*.

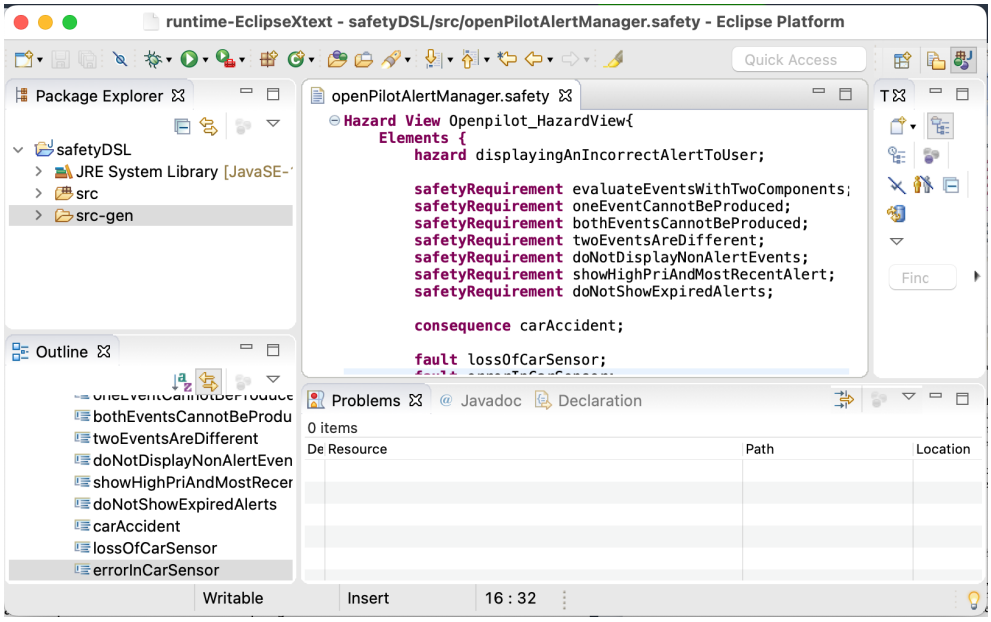

FIGURE 9 Tool for Safety DSL

7 | CASE STUDY EVALUATION

This section explains the application of our fault-based testing approach and presents the results by using an industrial case study described above. We applied the process shown in Figure 4. In the following, we explain the application of each step.

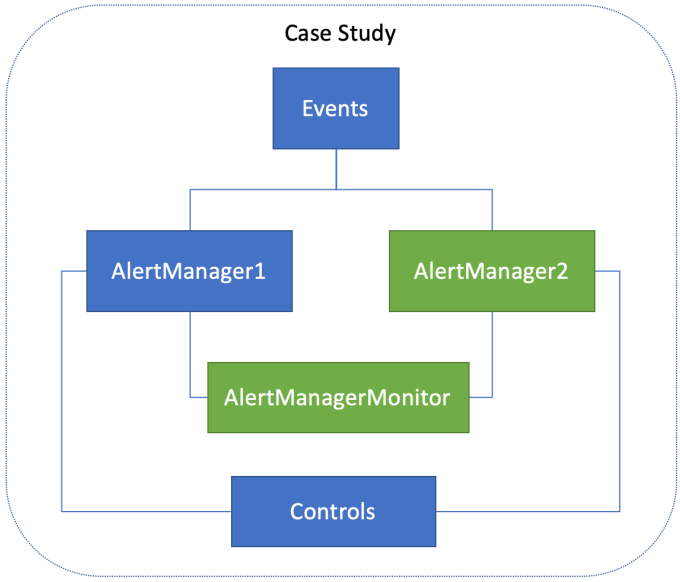

FIGURE 10 High-level architectural diagram of the case study

Component	Openpilot Module - Github Link
Events	events.py
Alert Manager1	alertmanager1.py
Alert Manager2	alertmanager2.py
Alert Manager Monitor	alertmanagermonitor.py
Controls	controlsd.py

TABLE 5 High-level architectural component along with Openpilot module

7.1 | Build Safety Model

As described in Section 3, we selected *displaying an alert to user when an unusual event occurs* hazard and identified safety requirements as shown in Table 2. Later, we defined the corresponding safety tactics and presented them in Table 3. In order to address the safety tactics we defined, we leveraged the part of Openpilot open-source software explained in Section 3 and built a case study. Figure 10 shows the overall high-level architecture diagram of our case study. We added another AlertManager (AlertManager2) to address Replication Redundancy defined in 3 for SR1. Table 5 presents the links to the Openpilot module associated with the component shown in Figure 10.

These are the first two steps of the proposed fault-based testing approach. As a next step, we built the safety model. For defining the safety-critical view, firstly, we identified our architectural elements. AlertManager1 and AlertManager2 shown in Figure 10 are responsible for processing alerts. Each AlertManager receives the alert data from

Controls. Controls read the alert data from car sensors using CAN protocol. If a warning or should be generated, AlertManagers notifies the Controls through commands relation. If a fault occurs in AlertManager1 and AlertManager2, they report the fault to Controls through reportsFault relation. For condition monitoring, voting, and recovery tactics, we added AlertManagerMonitor. AlertManagerMonitor monitors AlertManager1 and AlertManager2 components. It detects the failure when one of these managers fails and recovers from failures by stopping/starting/initializing the failed modules.

We built a safety model by using safety DSL explained in Section 4 according to the case description. We defined hazard view, safety tactic view, and safety-critical view. For the sake of simplicity, we present a small part of the code snippet from the safety-critical view in Figure 11. Hazard view and safety-tactic view are created similarly using the safety DSL.

Additionally, we defined implementation relations for the mutant generation and test case run steps. Module-Class Relations shows which safety-critical module consists of which implementation classes, Class-Test Case Relations shows which implementation class should be tested with which test case classes. We present the architecture to code relations in Figure 11. Architecture to code relations provides a mapping between the module defined in the architectural model and the class implemented in Java. Also, they map the implementation classes and test classes. For example, Figure 11 shows that *alertManager1* has the implementation in *alertManager.py* in module-class relations. In the next section, class-test case relations show that the test cases for *alertManager.py* is implemented in *test_alerts.py*, *test_state_machine.py*, *test_alertmanager.py*. The complete safety model for our case study can be found in [25].

7.2 | Identify and Create Mutants

In this step, we identified mutants based on the safety tactics that we implemented in our case study. Based on the mutation model we introduced in Table 4, we determined the mutants and presented them in Table 6. Table 4 explains the action item required to taken for each safety tactic. For example, for SR1, we have Replication Redundancy as a safety tactic in which we have AlertManager1 and AlertManager2 components as an application of this tactic. Based on the guideline we have in 4 we defined specific mutation models for each safety tactic we defined for the safety requirements.

We used the safety model and the selected mutation operators as inputs to our tool to create skeleton code for generating mutants for method-level mutation generation. The skeleton code includes the required Python code for mutant generation and execution of test cases. A sample code snippet is shown in below:

```
mut.py -t controlsd.py -u selfdrive/controls/tests/test_state_machine.py
       -o AOD AOR COD COI ROR LOR LOD --report-html Report-controlsd
```

This code snippet includes the mutant generation code for the *Sanity Check* tactic for the module Controls. The mutation operators that we have in the above code snippet are AOD (Arithmetic Operator Deletion), AOR (Arithmetic Operator Replacement), COI (Conditional Operator Insertion), COD (Conditional Operator Deletion), ROR (Relational Operator Replacement). LOR (Logical Operator Replacement), LOD (Logical Operator Deletion). COI and COD are works with conditional operators like && (and), || (or), & (bit-wise and), | (bit-wise or), ^ (xor), ! (not) where ROR is related to relational operators > (greater than), < (less than), >= (greater than or equals), <= (less than or equals), == (equals), != (not equals). All of the details on these operators can be found in [22].

For component-level mutation generation, we manually modified the code parts in the implementation of the case study to reflect mutations.

```
Safety-CriticalView Openpilot_Safety_Critical {
  Elements {
    monitor alertManagerMonitor {
      implementedTactics= healthCheckForAlertManager, recoverAlertManager
    };

    safety-critical alertManager1{
      criticalityLevel=B;
      implementedSafetyRequirements= showHighPriAndMostRecentAlert,
evaluateEventsWithTwoComponents;
      implementedTactics= checkAlertStates, compareAlertsTimeAndPriority, alertManagerReplica;
    };

    safety-critical alertManager2{
      criticalityLevel=B;
      implementedSafetyRequirements= showHighPriAndMostRecentAlert,
evaluateEventsWithTwoComponents;
      implementedTactics= checkAlertStates, compareAlertsTimeAndPriority, alertManagerReplica;
    };
    safety-critical controlsd {
      criticalityLevel=B;
      implementedSafetyRequirements= doNotShowExpiredAlerts, oneEventCannotBeProduced,
bothEventsCannotBeProduced, twoEventsAreDifferent;
      implementedTactics=checkIfAlertIsActive, alertManagerSelectAlert, compareAlerts,
alertManagerVoting;
    };
    safety-critical events {
      criticalityLevel=B;
      implementedSafetyRequirements= doNotDisplayNonAlertEvents;
      implementedTactics=eventTypeCheck;
    };
  }
  Relations {
    alertManagerMonitor monitors alertManager1, alertManager2;
    alertManagerMonitor stops alertManager1, alertManager2;
    alertManagerMonitor starts alertManager1, alertManager2;
    alertManagerMonitor inits alertManager1, alertManager2;

    controlsd reads alertManager1, alertManager2;
    controlsd commands alertManager1, alertManager2;
    alertManager1 reportsFault controlsd;
    alertManager2 reportsFault controlsd;
  }
}

ImplementationRelations {
  Module-Class Relations {
    alertManagerMonitor composesOf = { alertmanagermonitor };
    alertManager1 composesOf = { alertmanager };
    alertManager2 composesOf = { alertmanager2 };
    events composesOf = {events};
    controlsd composesOf = { controlsd };
  };
  Class-Test Case Relations {
    alertmanagermonitor testWith = { selfdrive.controls.lib.tests.test_alertmanagermonitor };
    alertmanager testWith = { selfdrive.controls.tests.test_alerts,
selfdrive.controls.tests.test_state_machine, selfdrive.controls.lib.tests.test_alertmanager };
    events testWith = { selfdrive.controls.lib.tests.test_alertmanager };
    controlsd testWith = { selfdrive.controls.tests.test_state_machine };
  };
```

FIGURE 11 Safety model definition for safety-critical view and architecture to code relations

Safety Requirement	Safety Tactic	Mutation Model
SR1	Replication Redundancy	Component level: Replace the implementation of alert managers such that they have different implementations.
SR2	Condition Monitoring Voting	Method level: Use mutation tool to generate mutations of corresponding code part for AlertManagerMonitor Component level: Create the mutations of the AlertManagerMonitor as to have faulty voting behavior
SR3	Condition Monitoring Repair	Method level: Use mutation tool to generate mutations of corresponding code part for AlertManagerMonitor Component level: Create the mutations of the AlertManagerMonitor as to have faulty repair behavior
SR4	Condition Monitoring Override	Method level: Use mutation tool to generate mutations of corresponding code part for Controls Component level: Create the mutations of the Controls as to have faulty override behavior
SR5	Sanity Check	Method level: Use mutation tool to generate mutations of corresponding code part for Events
SR6	Sanity Check Comparison	Method level: Use mutation tool to generate mutations of corresponding code part for Alert Managers
SR7	Sanity Check	Method level: Use mutation tool to generate mutations of corresponding code part for Controls

TABLE 6 Identifying Mutant Model for Case Study

	Lines of Mutated Code	Total Test Cases	Mutation Model	Total Generated Mutants	Equivalent Mutants	Alive Mutants	Killed Mutants	Mutation Score (%)
Alert Manager1	37	3	Replication Redundancy	4	0	0	4	100
Alert Manager2	37	3	Replication Redundancy	4	0	0	4	100

TABLE 7 Mutation Results for SR1

7.3 | Run Test Cases

The next step is executing test cases on mutant codes. Test case generation is performed by generated code for method-level mutants. As shown in Implementation Relations, *controlsd.py* should be tested with *test_state_machine* test class. For the sake of simplicity, the part of the generated code is shown in Figure **??**. As given in the code, test cases are executed on mutants for each test class, and results are collected to generate a report.

7.4 | Results

The last step is the generation of the report. When the test cases are executed in Section 7.3, results are collected, and the code part is called to generate a report. The report includes the classes under test, test case classes, mutation operators, test results (fail/pass), related faults, and related safety tactics. Tables 7-13 show the results for our case study along with the mutation score. They also include details of mutant generation (the total number of lines of the mutated code, mutation model, the total number of generated mutants, the total number of alive mutants, and the total number of killed mutants). While calculating the mutation score, equivalent mutants should be determined. If a mutant semantically behaves precisely like the original program, the mutant is equivalent. We manually checked the generated mutants to see if they behave like the original code to detect equivalent mutants. This process can be improved by using the existing approaches proposed in several studies, such as [26, 27, 28]. Since the original program passes all of the test cases and the killed mutant is a mutant that failed on at least one of the test cases, a killed mutant cannot be an equivalent mutant. In this regard, we only checked the live mutants to see if any of them is an equivalent mutant. We included the number of equivalent mutants as another column in the tables. We calculated the mutation score using the formula presented in Section 5.

Table 7, Table 10, Table 11 and Table 13 show the results for SR1, SR4, SR5 and SR7 respectively. For all of these, the mutation score is 100, no further action is required.

Table 8 presents the mutation results for SR2. For *Condition Monitoring and Voting* tactic, there are 10 mutants generated and 8 of them killed. When we revisited the test cases for Condition Monitoring tactic, we observed that some cases are not considered for this tactic. The test cases were missing some of the edge cases for checking the state of each AlertManager. We added three more test cases to cover all of the cases. With the complete test suite, all of the mutants were killed, and we obtained a mutation score of 100.

Table 9 presents the mutation results for SR3. For *Condition Monitoring and Repair* tactic, there are 11 mutants generated and 5 of them killed. When we revisited the test cases for Condition Monitoring tactic, we observed that some cases are not considered for the this tactic. When we revisited the test cases for this tactic, we observed that some cases are not considered for the Condition Monitoring tactic. The test cases were missing some of the edge

	Lines of Mutated Code	Total Test Cases	Mutation Model	Total Generated Mutants	Equivalent Mutants	Alive Mutants	Killed Mutants	Mutation Score (%)
Alert Manager Monitor	12	5	Condition Monitoring and Voting	10	0	2	8	80

TABLE 8 Mutation Results for SR2

	Lines of Mutated Code	Total Test Cases	Mutation Model	Total Generated Mutants	Equivalent Mutants	Alive Mutants	Killed Mutants	Mutation Score (%)
Alert Manager Monitor	19	4	Condition Monitoring and Repair	11	2	6	5	55.55

TABLE 9 Mutation Results for SR3

cases for checking the state of each AlertManager. We added two more test cases to cover all of the cases. With the complete test suite, all of the mutants were killed, and we obtained a mutation score of 100.

Table 12 presents the mutation results for SR6. For *Sanity Check and Repair* tactic, there are 11 mutants generated and 5 of them killed. When we revisited the test cases for Repair tactic, we observed that some cases are not considered for this tactic. The test cases were missing some of the edge cases for checking the state of each AlertManager. In first iteration, we added two more test cases to cover all of the cases. With the first iteration, we were able to kill 5 more mutants in which the mutation score is 55%. In the second iteration we added two more test cases and we were able to obtain the mutation score as 100%.

7.5 | Evaluation

In this section, we present the evaluation of our fault-based testing approach. We generated mutations for the implementation of the case study using the proposed approach and aim to achieve a 100% mutation score to locate the weaknesses in the test suite and have effective tests for safety concerns. For some of the safety requirements, the

	Lines of Mutated Code	Total Test Cases	Mutation Model	Total Generated Mutants	Equivalent Mutants	Alive Mutants	Killed Mutants	Mutation Score (%)
Controls	247	8	Condition Monitoring and Override	158	0	0	158	100

TABLE 10 Mutation Results for SR4

	Lines of Mutated Code	Total Test Cases	Mutation Model	Total Generated Mutants	Equivalent Mutants	Alive Mutants	Killed Mutants	Mutation Score (%)
Events	13	5	Sanity Check	45	0	0	45	100

TABLE 11 Mutation Results for SR5

	Lines of Mutated Code	Total Test Cases	Mutation Model	Total Generated Mutants	Equivalent Mutants	Alive Mutants	Killed Mutants	Mutation Score (%)
Alert Manager1	37	5	Sanity Check Comparison	21	1	14	6	30
Alert Manager2	37	5	Sanity Check Comparison	21	1	14	6	30

TABLE 12 Mutation Results for SR6

	Lines of Mutated Code	Total Test Cases	Mutation Model	Total Generated Mutants	Equivalent Mutants	Alive Mutants	Killed Mutants	Mutation Score (%)
Controls	342	8	Sanity Check	258	0	0	258	100

TABLE 13 Mutation Results for SR7

mutation score was already 100%. For these kinds of requirements, the implemented test suite is able to cover all the edge cases. However, for example, SR6 requires multiple safety tactics to be implemented, and these tactics have lots of edge cases to check. We revisited the test cases and observed that the test suite was missing some test cases.

With the help of our approach, engineers or developers build system models by focusing on safety concerns. They use these models as an input to mutation testing and evaluate the adequacy of the test suite based on the safety concerns explicitly defined in the models. If the mutation score is not 100, they revisit and reiterate the test suite to add missing or edge test cases to achieve a mutation score of 100. Our tool automates the process by removing manual steps for generating the mutations and running test cases. It also helps to ensure the safety concerns are properly addressed in the test cases by focusing on the safety tactics.

8 | DSL EVALUATION

In this section, we present the evaluation of our DSL from the end users' perspective. Since our DSL is relatively new, we do not have adequate trained users to conduct formal interviews with questionnaires to evaluate our DSL. In this regard, we have looked at the existing studies in the literature to provide an approach for assessing DSLs from various perspectives. [29, 30, 31, 32] propose different approaches to evaluate novel DSLs. For our DSL, we used Framework for Qualitative Assessment of DSLs (FQAD) [30], which is based on the ISO/IEC 25010:2011 standard. FQAD describes a set of quality properties for assessing a DSL, including Functional suitability, Usability, Reliability, Maintainability, Productivity, Extensibility, Compatibility, Expressiveness, Reusability, and Integrability. In the following, we present the evaluation of our DSL considering each quality characteristic.

Functional suitability indicates to what degree the DSL is fully developed. This means that all necessary functionalities exist in the DSL, and the DSL does not have functionality not given in the represented domain. We used our DSL to define multiple case studies, and we have been able to describe all the problem-specific functionalities needed to express safety. From this point, we can conclude that our DSL meets this criterion.

Usability refers to the degree to which specified users can use DSL to accomplish specified goals. To analyze this property, we have asked engineers experienced in the safety domain to assess the overall usability of our DSL. They expressed that differentiating between safety-critical and non-safety-critical components in the system helped them identify where to focus on the safety requirements. They also indicated that expressing and seeing the direct relation between safety-critical components and safety tactics helped address safety concerns in the system. Overall, they mentioned that the DSL is easy to learn and use.

Reliability of a DSL is defined as the property of a language that helps to produce reliable programs. We developed our DSL using the Xtext framework in the Eclipse environment. The Xtext framework provides full infrastructure including parser, linker, type checker, compiler and editing support for Eclipse. The Eclipse editor provides all the requirements for handling code errors.

Maintainability shows to what degree the DSL is easy to maintain. Our DSL consists of four main parts, which are defined by applying the separation of concerns principle. This helps to achieve modularity in the DSL. For maintainability, it is also vital to address understandability. In our DSL, we directly model the concepts as defined in the safety domain. Therefore, the grammar is easy to understand. Maintenance also covers modifiability. Since our DSL design is modular, it can be easily modified, or new concepts can be added.

Productivity refers to the degree to which a DSL promotes programming productivity. Our DSL helps to increase productivity because it enhances the design and testing process of safety-critical systems. It helps developers and engineers to identify safety-critical concerns by explicitly defining them at the early stages of the design. Also, it

supports the testing stage by helping test engineers to assess the quality of the test suites focusing on the safety concerns.

Extensibility defines the degree to which a DSL has general mechanisms for users to add new features. Our DSL can easily be extended because of its modularity. Our DSL consists of four different parts that each provides different a viewpoint to the safety domain. In this regard, our DSL can be easily extended by adding new concepts. Also, the Xtext framework and the Eclipse helps users to add new features to DSL easily.

Compatibility of a DSL shows at what degree a DSL is compatible with the domain and the development process. We defined our DSL to enhance the testing process of safety-critical systems. It is designed to help test engineers to assess the test suites' quality by focusing on safety concerns. It fits the systems engineering lifecycle in terms of requirement analysis, design, development, and testing.

Expressiveness defines the relation between the program and what the programmer has in their mind. For this criterion, it is imperative to have a one-to-one mapping between the concepts and their representation in the DSL. We developed our DSL based on a thorough domain analysis whereby we have modeled each concept in the corresponding metamodel of the language. We can affirm a one-to-one correspondence between the concepts, and their representation in the DSL and there are no duplicated concepts. We also considered the abstraction level of the concepts in the DSL to ensure that they are not too generic or too specific but expressive enough to represent the safety domain.

Reusability of a DSL refers to the degree to which DSL can be used in any other language. The definitions in our DSL can be used in any other language since the DSL directly models the concepts as defined in the safety domain.

Integrability defines the degree to which the DSL is compatible with integration with other languages. We developed the DSL using the Xtext framework in Eclipse environment. The Eclipse platform allows developers to extend Eclipse applications like Eclipse IDE with additional functionalities via Eclipse plug-ins. In this respect, our DSL can be integrated with other languages using the Eclipse IDE.

9 | RELATED WORK

Several studies have proposed domain-specific languages (DSL) for addressing safety. In [33], the authors define a DSL to present Petri-nets and a tool MeeNET to debug safety-critical systems. Their focus is on having a DSL to formally define behavior of the system using Petri-nets and verify the system behavior. Nandi et al. [34] proposes a DSLfor the correct deployment of RV solutions in the scope of cyber-physical systems. Kaleeswaran et al. [35] define a DSL for Hazard and Operability Analysis (HAZOP) study. HAZOP study is a systematic way to identify potential hazards in the system. The HAZOP-DSL helps users to build links between HAZOP study and the system model for consistency and traceability. With the DSL support, the changes in the system model can be detected, and the user is enforced to make necessary changes in the HAZOP. Their study enables users to detect issues in the safety analysis at early design levels. [36] defines a DSL for defining the safety requirements and automatically verifying their consistency using formal methods. They propose a domain-specific language SafeNL to enable users to define safety requirements formally. They convert the SafeNL documents into formal constraints (Clock Constraint Specification Language) and verify their consistency with existing tools. Queiroz et al. [37] propose a DSL for defining scenarios used in simulation testing of autonomous driving systems. Their DSL includes information about vehicles, pedestrians, paths, roads which are the main components to compose test cases for self-driving vehicle testing. Some studies [38, 39] define a DSL to improve processes in their engineering life cycle. [39] defines a DSL (Mauve) for specifying the software architecture of autonomous robots. Using this DSL, they analyze the real-time correctness of the architecture by

verifying the schedulability of different components. They transform Mauve model into Periodic State Machines and analyze real-time characteristics of the architectural components. Also, they check the validity of behavioral properties by converting Mavue model into Fiarce [40] models and analyze the Fiarce model using TINA [41]. In [42], the authors propose a domain-specific modeling language (DSML) to provide a conceptual model for expressing the information mandated by DO-178C standard. Iber et al. [43] proposes a DSL to specify tests from UML Testing Profile (UTP). They model the UTP to support model-driven development processes such as generating test code.

All these and other studies in the literature define DSL to support system development lifecycle. Some focus only on one step of the lifecycle such as requirement analysis or design or implementation, where some of them are specific to one safety-critical domain like railway systems,robotics, or automotive. To the best of our knowledge, no generic DSL has been presented which is dedicated to assess the quality of the test suite. With this study, our primary focus is evaluating the adequacy of the test suite based on the applied safety tactics in the software architecture models. Our DSL allows us to express safety tactics in the safety models and the system implementation details to generate mutants and run test cases.

In the literature, several studies proposed a fault-based testing approach to test safety-critical systems. In [44], the authors proposes an approach to generate test oracles from the formal requirements defined in CASDL (Casco Accurate Specification Description Language). In another study [12], a test case generation approach is defined based on model mutation for the safety requirements in the system. Firstly, a fault model is defined by describing mutation operators and UML models of the system. Then, they define a process for transforming a UML model to OOAS (Object-Oriented Action Systems) using fault models. Subsequently, OOAS models' mutations are generated and used for the test case generation process. Another study [45] applies mutation testing on a nuclear reactor. In this work, a test case generation approach is defined to test a nuclear reactor. Mutation testing is applied by mutating the source code. With this approach, they aim to calculate the degree of test adequacy of the generated test cases.

Safety concern has not been explicitly addressed using a dedicated architecture perspective before. However, there is plenty of work related to safety engineering. In our earlier work [19, 46], we have provided a safety perspective that can support the architectural design of safety-critical systems. It can assist the system and software architects in designing, analyzing, and communicating the decisions regarding safety concerns by evaluating safety issues early on the life cycle before implementing the system.

In [47, 48], several architectural patterns are proposed to support software safety design. Gawand et al. [48] propose a framework for the specification of architectural patterns to support safety and fault tolerance. They provide four types of patterns. One of the patterns is Control-Monitor pattern. They aim to improve fault detection by using redundancy by using this pattern. Another pattern is the Triple Modular Redundancy pattern which is used to enhance system's safety where there is no fail-safe state. The other pattern is the Reflective State pattern which separates the application into base-level and meta-level to separate control and safety aspects from the application logic. The last pattern is Fault Tolerance Redundancy pattern which improves the fault tolerance of the system while implementing the redundancy for safety. Armoush et al. [47] propose a Recovery Block with Backup Voting pattern which improves the fault tolerance of the system.

Our earlier work considered the explicit modeling of viewpoints for quality concerns [49, 50, 51]. As a result, each quality concern, such as adaptability and recoverability, requires a different decomposition of the architecture. Architectural elements and relations are defined to specify the required decomposition for the quality concerns. Earlier work on local recoverability has shown that this approach is also broadly applicable. We consider this work complementary to the architectural perspectives approach. Both alternative approaches seem to have merits.

10 | THREATS TO VALIDITY

In this section, we discuss threats to the validity of our study using the guideline defined in [52].

Construct validity: The main goal of our study is to assess the adequacy of the test suite of safety-critical systems. To achieve this, we built an approach by leveraging existing fault-based testing methods. While applying the proposed approach to the case study, we assumed that the implementation (code) of the case study is bug-free and the test suite is complete from a test coverage perspective. Any defects in the case study implementation and test suite may affect our case study evaluation results. Additionally, we used μJava [53] to generate mutants. Any issues in μJava would jeopardize our study's construct validity. The other point we bring to attention is that our case study is implemented in Java. We did not focus on the requirements of programming languages for supporting the implementation of safety-critical concerns since the scope of the paper is on mutant generation. This also might have an effect on the construct validity.

Internal validity: To evaluate our approach, we used a use case from a real industrial case study. For the given case study, there is no equivalent mutant detected. Some of the equivalent mutants could not be generated or found because of the size of the case study. This might cause a threat to our internal validity. We plan to perform additional case studies as future work.

External validity: Our approach is based on safety concerns, and it provides a generic approach for safety-critical systems. To illustrate our work, we applied our approach to a real case from a safety-critical system in the avionics domain. However, it can be applied to any safety-critical system from any domain since the overall approach is generic.

Reliability: In this work, we provided detailed information about each step of the proposed approach. The meta-model, DSL, and tool are publicly available through an open-source platform. Hence, the results obtained within our study are reproducible.

11 | CONCLUSION

Testing safety-critical systems is essential and for this purpose, developing an effective test suite is necessary. In this article, we have thus provided a systematic approach for assessing test suites of safety-critical systems. For this purpose, we have adopted a fault-based testing approach that can be used to analyze the effectiveness of so-called architecture safety tactics. We have developed the required metamodel and realized the DSL to model the faults and tactics and support fault-based testing. We have applied the approach and the tool for a real industrial case study. The approach and the tool are helpful to assess a given test suite and analyze the strength of the safety tactics.

Based on the results from our case study, our main conclusion is that our approach is feasible and effective for test suite assessment of safety-critical systems. It supports the overall architecture design of safety-critical systems and analysis to realize the requirements for safety-critical systems. With our fault-based testing approach, engineers and developers build dedicated system models to express safety concerns, use these models as an input to mutation testing, and evaluate the effectiveness of the developed test suite based on the safety concerns addressed in the models. If the mutation score is not 100, they revisit and reiterate the test suite to add missing or edge test cases to achieve a mutation score of 100. Since our approach focuses on safety tactics and fault knowledge, it enables developers to build complete and robust test suites focusing on safety concerns while building safety-critical systems.

For future developments, we aim to enhance the approach further by systematically analyzing different faults and safety tactics from various domains such as robotics, nuclear systems, and automotive. Besides, we aim to automate mutation operation selection with the help of the safety model. Another improvement area that we plan is generating

test artifacts (test data, test scripts, test oracle) from the DSL we defined. We also consider adding debugging and testing support to our DSL as future work.

Acknowledgements

Authors thank their universities for providing the infrastructure.

Conflict of interest

The authors declare no conflict of interest.

Supporting Information

There is no supporting information.

references

[1] Leveson N, Harvey P. Analyzing Software Safety. IEEE Transactions on Software Engineering 1983 sep;9(05):569–579.

[2] Ericson C. In: Hazard Analysis Techniques for System Safety John Wiley & Sons, Ltd; 2005. .

[3] Sparkman D. Techniques, processes, and measures for software safety and reliability. Nuclear Systems Safety Program 1992;.

[4] Leveson NG. Safeware: System Safety and Computers. New York, NY, USA: ACM; 1995.

[5] NASA Software Safety Guidebook;. URL: `https://standards.nasa.gov/standard/nasa/nasa-gb-871913`.

[6] Rozanski N, Woods E. Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives. Addison-Wesley Professional; 2005.

[7] Wu W, Kelly T. Safety tactics for software architecture design. In: Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.; 2004. p. 368–375 vol.1.

[8] Gurbuz HG, Pala Er N, Tekinerdogan B. Architecture Framework for Software Safety. In: System Analysis and Modeling: Models and Reusability Cham: Springer International Publishing; 2014. p. 64–79.

[9] Xue-Fang D, Rui Z. A scenario-based lightweight software architecture analysis method. In: 3rd International Conference on Green Communications and Networks, vol. 54 VOLUME 1; 2014. p. 949–956.

[10] Buchgeher G, Weinreich R. An approach for combining model-based and scenario-based software architecture analysis. In: 2010 Fifth International Conference on Software Engineering Advances; 2010. p. 141–148.

[11] Tekinerdogan B, Sozer H, Aksit M. Software architecture reliability analysis using failure scenarios. Journal of Systems and Software 2008;81(4):558–575.

[12] Herzner W, Schlick R, Brandl H, Wiessalla J. Towards Fault-based Generation of Test Cases for Dependable Embedded Software. Softwaretechnik-Trends 2011;31.

[13] Preschern C, Kajtazovic N, Kreiner C, et al. Catalog of safety tactics in the light of the IEC 61508 safety lifecycle. In: Proceedings of VikingPLoP 2013 Conference; 2013. p. 79.

[14] Pezze M, Young M. Software Testing and Analysis: Process, Principles, and Techniques. Wiley; 2007.

[15] Ammann P, Offutt J. Introduction to Software Testing. 1 ed. USA: Cambridge University Press; 2008.

[16] Openpilot: an open-source driver assistance system.;. URL: https://github.com/commaai/openpilot.

[17] Galanopoulou R, Spinellis DD. A Dataset of Open-Source Safety-Critical Software; 2021. .

[18] Openpilot high-level component diagram.;. URL: https://github.com/commaai/openpilot/wiki/Introduction-to-openpilot#inter-process-communication.

[19] Gurbuz HG, Tekinerdogan B, Pala Er N. Safety Perspective for Supporting Architectural Design of Safety-Critical Systems. In: Software Architecture Cham: Springer International Publishing; 2014. p. 365–373.

[20] International Organization for Standardization, editor, ISO/IEC 14977:1996 Information Technology - Syntactic Metalanguage - Extended BNF; 1996.

[21] [MIL-STD-882D]. Standard Practice for System Safety, Department of Defense; 2000.

[22] MutPy, Mutation testing tool for Python 3.3;. URL: https://github.com/mutpy/mutpy.

[23] Xtext Home Page;. URL: https://www.eclipse.org/Xtext.

[24] Xtend Home Page;. URL: https://www.eclipse.org/xtend.

[25] Safety DSL tool;. URL: http://github.com/havvagulay/safetyDsl.

[26] Offutt AJ, Jie Pan. Detecting equivalent mutants and the feasible path problem. In: Proceedings of 11th Annual Conference on Computer Assurance. COMPASS '96; 1996. p. 224–236.

[27] Madeyski L, Orzeszyna W, Torkar R, Józala M. Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. IEEE Transactions on Software Engineering 2014 Jan;40(1):23–42.

[28] Adamopoulos K, Harman M, Hierons R. How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution. In: Genetic and Evolutionary Computation – GECCO 2004, vol. 3103; 2004. p. 1338–1349.

[29] Mernik M, Heering J, Sloane A. When and How to Develop Domain-Specific Languages. ACM Comput Surv 2005 12;37:316–.

[30] Kahraman G, Bilgen S. A framework for qualitative assessment of domain-specific languages. Software & Systems Modeling 2015 Oct;14(4):1505–1526. https://doi.org/10.1007/s10270-013-0387-8.

[31] Oliveira N, Pereira MJV, Henriques PR, da Cruz DC. Domain specific languages: a theoretical survey. In: INForum'09-Simposio de Informatica; 2009. .

[32] Kosar T, Oliveira N, Mernik M, João M, Pereira M, Repinšek M, et al. Comparing General-Purpose and Domain-Specific Languages: An Empirical Study. Computer Science and Information Systems 2010 05;438.

[33] Idani A. Formal model-driven executable DSLs. Innovations in Systems and Software Engineering 2021;.

[34] Nandi GS, Pereira D, Proença J, Tovar E. Work-In-Progress: a DSL for the safe deployment of Runtime Monitors in Cyber-Physical Systems. In: 2020 IEEE Real-Time Systems Symposium (RTSS); 2020. p. 395–398.

[35] Kaleeswaran AP, Munk P, Sarkic S, Vogel T, Nordmann A. A Domain Specific Language to Support HAZOP Studies of SysML Models. In: Papadopoulos Y, Aslansefat K, Katsaros P, Bozzano M, editors. Model-Based Safety and Assessment Cham: Springer International Publishing; 2019. p. 47–62.

[36] Chen X, Zhong Z, Jin Z, Zhang M, Li T, Chen X, et al. Automating Consistency Verification of Safety Requirements for Railway Interlocking Systems. In: 2019 IEEE 27th International Requirements Engineering Conference (RE); 2019. p. 308–318.

[37] Queiroz R, Berger T, Czarnecki K. GeoScenario: An Open DSL for Autonomous Driving Scenario Representation. In: 2019 IEEE Intelligent Vehicles Symposium (IV); 2019. p. 287–294.

[38] Anderson M, Bowman J, Kilgo P. RDIS: Generalizing domain concepts to specify device to framework mappings. In: 2012 IEEE International Conference on Robotics and Automation; 2012. p. 1836–1841.

[39] Gobillot N, Lesire C, Doose D. A Modeling Framework for Software Architecture Specification and Validation. In: Simulation, Modeling, and Programming for Autonomous Robots; 2014. p. 303–314.

[40] Berthomieu B, Bodeveix Jp, Farail P, Filali M, Garavel H, Gaufillet P, et al. Fiacre: an Intermediate Language for Model Verification in the Topcased Environment. Proc of the Embedded Real Time Software (ERTS) 2008 01;.

[41] Berthomieu B, Vernadat F. Time Petri nets analysis with TINA; 2006. p. 123–124.

[42] Metayer N, Paz A, El Boussaidi G. Modelling DO-178C Assurance Needs: A Design Assurance Level-Sensitive DSL. In: 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW); 2019. p. 338–345.

[43] Iber J, Kajtazović N, Höller A, Rauter T, Kreiner C. Ubtl UML testing profile based testing language. In: 2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD); 2015. p. 1–12.

[44] Zheng H, Feng J, Miao W, Pu G. Generating Test Cases from Requirements: A Case Study in Railway Control System Domain. In: 2021 International Symposium on Theoretical Aspects of Software Engineering (TASE); 2021. p. 183–190.

[45] Babu PA, Kumar CS, Murali N, Jayakumar T. An Intuitive Approach to Determine Test Adequacy in Safety-critical Software. SIGSOFT Softw Eng Notes 2012 Sep;37(5):1–10.

[46] Gurbuz HG, Tekinerdogan B. Model-based testing for software safety: a systematic mapping study. Software Quality Journal 2018 Dec;26(4):1327–1372.

[47] Armoush A, Salewski F, Kowalewski S. Recovery Block with Backup Voting: A New Pattern with Extended Representation for Safety Critical Embedded Systems. In: 2008 International Conference on Information Technology; 2008. p. 232–237.

[48] Gawand H, S Mundada R, Swaminathan P. Design Patterns to Implement Safety and Fault Tolerance. International Journal of Computer Applications 2011 03;18.

[49] Sozer H, Tekinerdogan B. Introducing Recovery Style for Modeling and Analyzing System Recovery. In: Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008); 2008. p. 167–176.

[50] Sozer H, Tekinerdoğan B, Akşit M. Optimizing decomposition of software architecture for local recovery. Software Quality Journal 2013 Jun;21(2):203–240. https://doi.org/10.1007/s11219-011-9171-6.

[51] Tekinerdogan B, Sozer H. Defining Architectural Viewpoints for Quality Concerns. In: Software Architecture, vol. 6903; 2011. p. 26–34.

[52] Runeson P, Höst M. Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering 2008;14(2):131.

[53] muJava Home Page; 2014. URL: http://cs.gmu.edu/~offutt/mujava.

A | EBNF GRAMMAR OF DSL

```
SafetyDSL  = {SafetyView} ImplementationDetail;
SafetyView  = HazardView | SafetyTacticView | SafetyCriticalView;

HazardView = 'Hazard View' STRING '{ Elements {' {HazardElement} '}' 'Relations {' {HazardRelation} '}' '}' ;
HazardElement  =Hazard | SafetyRequirement | Consequence | Fault  | FaultTree ;
Hazard = 'hazard' HazardID ';' ;
SafetyRequirement = 'safetyRequirement' SReqID ';' ['{' {SafetyRequirement} '}'] ;
Consequence = 'consequence' ConsequenceID ';' ;
Fault = 'fault' FaultID ';' ;
FaultTree = 'faultTree' FaultTreeID FaultTreeNode ';' ;
FaultTreeNode = FaultID | ANDNode | ORNode ;
ANDNode = FaultTreeNode 'AND' FaultTreeNode ;
ORNode = FaultTreeNode 'OR' FaultTreeNode
HazardRelation = DerivedFrom | Causes | CausedBy ;
DerivedFrom = SReqID {',' SReqID} 'derivedFrom' HazardID  ';' ;
Causes = HazardID 'causes' ConsequenceID {',' ConsequenceID}  ';' ;
CausedBy = HazardID 'causedBy' FaultTreeID ';' ;

SafetyTacticView= 'SafetyTacticView' STRING '{' {SafetyTactic} '}' ;
SafetyTactic = ('faultAvoidance' | 'faultDetection' | 'faultContainment') STacticID '{' 'type=' STRING 'handledFaults=' (FaultID) {',' FaultID} '}' ';' ;

SafetyCriticalView = 'Safety-CriticalView' name=ID '{' 'Elements {'{ArchitecturalElement} '}' 'Relations {'{SafetyCriticalRelation} '}' '}' ;
ArchitecturalElement = SafetyCritical | NonSafetyCritical | Monitor ;
SafetyCritical = 'safety-critical' SCModuleID '{' 'criticalityLevel=' ('A' | 'B' | 'C' | 'D') ';'
                 'implementedSafetyRequirements=' SReqID {',' SReqID} ';'
                 ['implementedTactics=' STacticID {',' STacticID} ';' ]
                 ['sub-elements='{SCModuleID} {',' SCModuleID} ';' ]
                 ['hasState' StateID {',' StateID}] '}' ';' ;
NonSafetyCritical = 'non-safety-critical' NSCModuleID ( '{' {NSCModuleID} '}' | ';' ) ;
Monitor = 'monitor' MonitorID ['{' 'implementedTactics=' SCTacticID {',' SCTacticID} '}'] ';' ;
State = ('state' | 'safeState') StateID  ';' ;

SafetyCriticalRelation = ArchElementToArchElement | MonitorToArchitecturalElement | ReportsFault ;
ArchitecuralElementID = (SCModuleID | NSCModuleID | MonitorID);
ArchElementToArchElement = ArchitecturalElementID ('reads' | 'writes' | 'commands') ArchitecturalElementID {',' ArchitecturalElementID} ';' ;
MonitorToArchitecturalElement = MonitorID ('stops' | 'starts' | 'inits' | 'restarts' | 'monitors') SCModuleID {';' SCModuleID} ';' ;
ReportsFault = SCModuleID 'reportsFault' SCModuleID {',' SCModuleID} ';' ;
ImplementationDetail = 'ImplementationRelations {' 'Module-Class Relation {' {ModuleClassRelation} '};'
                       'Class-TestCase Relation{'  {ClassTestCaseRelation}  '};'     '}' ;
ModuleClassRelation = SCModuleID 'composesOf=' '{' ClassDef {',' ClassDef} '}' ';' ;
TacticTestCaseRelation = STacticID  'testWith=' '{'  QualifiedName {',' QualifiedName} '}' ';' ;
ClassDef = QualifiedName ;                                  FaultTreeID = STRING;
Qualified Name = STRING { '.' STRING } ;                    SCModuleID = STRING;
HazardID = STRING;                                          NSCModuleID = STRING;
SReqID = STRING;                                            MonitorID = STRING;
ConsequenceID = STRING;                                     StateID = STRING;
FaultID = STRING;                                           STacticID = STRING;
```