Adaptation of an auto-generated code using a model-based approach to verify functional safety in real scenarios

Joelle Abou Faysal, Nour Zalmai, Ankica Barisic, Frédéric Mallet

Adaptation of an auto-generated code using a model-based approach to verify functional safety in real scenarios

Joelle Abou Faysal*†, Nour Zalmai†, Ankica Barišić*, Frederic Mallet *
*Universite Cote d'Azur, Cnrs, Inria, I3S, Sophia Antipolis, France
†Renault Software Factory, Sophia Antipolis, France
Email: joelle.abou-faysal@etu.univ-cotedazur.fr, nour.zalmai@renault.com,
Ankica.Barisic@univ-cotedazur.fr, Frederic.Mallet@univ-cotedazur.fr

***Abstract*—The level of autonomy of our vehicles is rapidly increasing. However, the acceptance of fully Autonomous Vehicles (AVs) depends on the confidence in their ability to operate safely in an uncontrolled environment. Hence, experts and non-experts must have a rigorous method along with adequate tools that can support their exigencies and safety specifications. This paper presents a Domain-Specific Modeling Language (DSML) for defining formal rules and generating flaw-less artefacts, which enables the application of a Safety Analysis of Violations and Inconsistencies (SAVI). The validity of the approach is illustrated on a Renault use case implementation with formal safety goals for autonomous vehicles. Our approach allows designers to detect violation ambiguities and rule inconsistencies on real or simulated scenarios.**

***Index Terms*—Autonomous vehicles, safety rules, model-based system engineering, formal methods, requirement engineering, model development and verification, test and simulation.**

Almost every mode of transportation is becoming autonomous. The main difficult hurdle in the autonomous domain is to guarantee that systems and software components are safe. The automotive industry is investing a lot in deploying self-driving systems in transportation technologies. It is necessary to overcome these challenges before having big fleets of cars on our roads. To avoid the rejection from the public opinion, we need to get them involved in the adoption of safety decisions. Operational safety defined in the ISO 26262 standard [1] implies having a design for the safety component that deals with all unsafe situations. The safety of intended functionality (SOTIF) approach that extends ISO 26262, is defined in ISOPAR 21448.1. SOTIF is concerned with failure causes related to system performance limitations and predictable misuse of the system. Performance limitations or insufficiency of the implemented functions are due to technical limitations such as sensor performance and noise. They can also be due to limitations of the algorithm such as object detection failures and limitations of actuator technology. Safety experts started to use traditional manual approaches to enforce safety decisions [2]. There is a threat of using these approaches, as safety experts' analyses depend on their experiences. Sometimes safety rules are not well formalized and usually are not reusable in the future. In addition, classical exhaustive verification techniques cannot guarantee that the system is safe because of the high degree of uncertainty in the environment. These test-based approaches also lead to a system complexity where time and cost are not under control [3]. Using the Model-Based System Engineering (MBSE) approach to assess software safety, enables formalization, improves reuse of the software, and helps to address safety analysis [4]. Model-driven approaches also address the complexity with a model-centric methodology that exploits domain models rather than documents. The use of Domain-Specific Modeling Language (DSML) enables fast prototyping of those behaviors by using a metamodeling structure. [5]. Endowed with formal semantics it brings a possibility to verify before generating a conforming code.

In this paper, we propose a DSML to verify the safety of Autonomous Vehicles (AVs). Monitors are generated to ensure the functional safety. We apply the study to real scenarios where we visualize many types of breaches.

This paper is organized as follows. Section I presents relevant background information. Section II presents the DSML proposed. We begin by detailing the language development and presenting the user process where he is informed of the procedure to deploy this language. The last part of this section describes our Safety Analysis of Violations and Inconsistencies (SAVI) on Renault use case. Finally, section III concludes and discusses future directions.

I. Background

The design of safety-critical systems involves people with different expertise. All DSML users, whatever their domain of expertise, must correctly evaluate the architecture and understand the impact of their decisions on safety. On the other hand, safety experts also need solutions to ensure a good coverage in the considered safety scenarios and use cases. Fortunately, DSML bridge the gap by translating artefacts from one domain to another, and maintaining a full synchronization with safety models [6]. The certification of the generation process from models to deployed artefacts ensures that designers can safely focus on models, less on implementation details, and therefore reduce development time. Thus, it improves the efficiency of the testing process, almost reduced to integration,

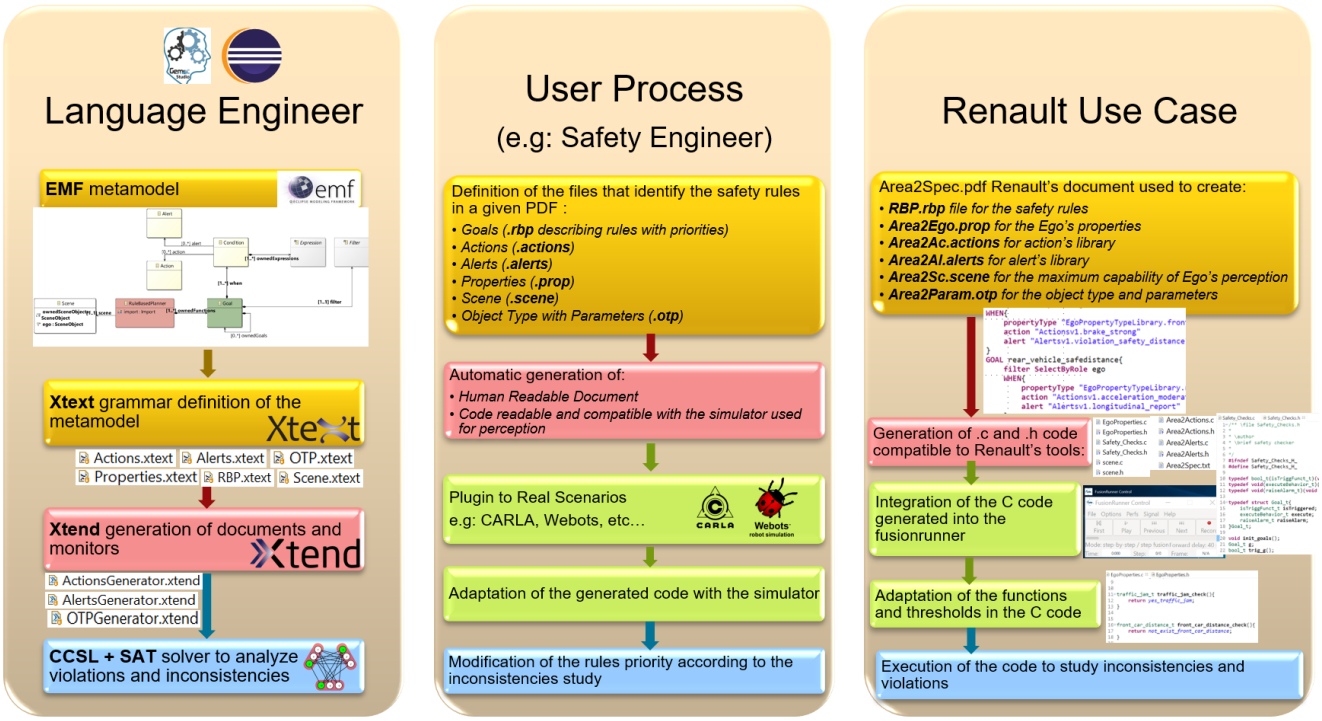


Fig. 1: Three-way views of the approach

as only safe artefacts are produced. Generated artefacts will then be used by engineers to provide a verification failure assessment.

Many approaches use the Model-Based System Engineering (MBSE) approach to provide a verification failure assessment. A safe autonomous vehicle trajectory DSML consists of driving a vehicle through a set of known waypoints by connecting motions in a sequence [6]. This work provides always a safe trajectory and creates a motion navigator for the autonomous car. Unfortunately, their domain is not applicable if the trajectory is already planned. It also does not provide to the user the alerts and the measures to take during an unsafe scenario, and does not know the original cause of the violation. An intelligible model is done to guarantee safety [7], but it is parametric and depends on environmental conditions. Sensor uncertainty dilemma has been handled using probabilistic models with formal specifications [8]. A rule-based strategy was also designed to evaluate sensors' dependability [9]. Yet, the main problem of autonomous driving systems perception is still not explored. In this paper, we give the user the capability to add parameters and specifications relying on the sensors and the environment, to examine ambiguities. Furthermore, a SceML study was carried out [10] to create a graphical model to generate new scenarios or test existing ones using Machine Learning. They facilitate the scenario creation process, but not the formal description of safety rules. This is why we apply formal semantics to this approach that allows specifying, designing, and analyzing the system. Mathematical reasoning can improve software reliability and dependability and is essential when developing complex software systems.

We develop an abstract model expressive enough to support safety analyses. We rely on the GeMoC framework [11] as it integrates a set of languages and tools based on the Eclipse Modeling Framework (EMF) to ease the definition of new DSMLs with inherent concurrency [12]. Additional components may be included to achieve artefact generation, such as correct syntactical code generation [13].

II. A Domain-Specific Modeling Language for the Safety of Autonomous Vehicles

We have developed a Domain-Specific Modeling Language (DSML) to verify the safety of Autonomous Vehicles (AVs). To ensure the acceptability of this language, we need to support the specification of the formal safety rules and the environment. An automatic generation of a monitoring system assists the interpreter by triggering alarms and possible recover maneuvers. Those monitors are expected to enable the user to detect inconsistencies and violations between rules and priorities. After performing a Safety Analysis of Violations and Inconsistencies (SAVI), the user can modify the rules and repeat the same cycle to ensure the correctness of the autonomous behavior.

We present three perspectives of our approach as seen in Fig. 1. First, from a language engineer perspective in section II-A, we describe necessary steps to implement the Extensible Platform for Safety Analysis of Autonomous Vehicles (EPSAAV) in [14]. EPSAAV is our DSML specified in our previous study where we detailed our metamodel and concrete syntax to auto-generate monitors. Second, we introduce gen-

eral steps for user perspective in section II-B. In this section, we detail how the user should use our approach and what he needs to do to perform a safety assessment. The third point of view describes Renault's use case in section II-C to test the approach and validate a SAVI in section II-C4.

A. Language Development

One of the main objectives of our language is to empower designers and experts with formal and yet practical solutions to describe the environment, the expected behavior, and the safety rules for the car under design. This is described in sections II-A1 and II-A2.
The second objective is to support the automatic generation of : (a) a human-readable document describing the rules and libraries used, and (b) monitors that allow the user to adapt output data of the simulator used with our safety rules. The violation of safety rules triggers alarms through these monitors. Section II-A3 details this part.
The third objective described in section II-A4, is to test, verify and identify inconsistencies. Sometimes a rule can lead to similar or contradictory behavior to another one. To avoid this, an inconsistency study must be conducted.
We use the open-source Gemoc tool [11] as it covers all aspects of a DSML development; from abstract and concrete syntax definition to semantics and operations. Gemoc is easy to integrate with all those technologies. It integrates solutions to ease the code generation and includes solutions to describe concurrent behaviors [12].

1) Abstract syntax: contains a graphical description of the metamodel. We use Eclipse Modeling Framework (EMF) technology. EMF is a framework and code generation facility that defines the model and generates implementation classes. EMF unifies the three important technologies: Java, XML, and UML. EMF model is the common high-level representation that glues them all together. The metamodel describes the classes and the relationships of the environment. For example, Fig.2 describes the abstract part of the rule-based planner, where *goals* are specified and composed of *conditions* referred to *alerts* and *actions*. The *goals* could be filtered either by role or by expression. The *RuleBasedPlanner* refers to a *Scene* that captures the perception capability of the vehicle. Rules are combined with logical operators. It is also important to prioritize rules in the case where several of them can be simultaneously triggered with contradictory behavior (e.g. break hard vs. maintain speed). The notion of *SelectByGoal* allows executing goals either in parallel or sequentially. Once the abstract syntax is specified, we build a concrete syntax.

2) Concrete syntax: defines the concrete terms that should be used by designers and the grammatical rules to bind them. We use Xtext technology to provide a concrete textual syntax to our language. In our language, we create Xtext files for some of the classes as defined in Fig.3. We get a separate description for the *scene*, *actions* and *alerts*, *parameters* and *properties*, and *goals* and priorities. The extensions defined for each class serve to create new files and libraries.

3) Auto-generation of a human-readable document and monitors: which enables safety engineers to easily integrate them with the chosen simulator and adapt the auto-generated code to check violations and consistencies described in section II-A4. We use Xtend technology to give the operational semantics and assign a behavior to each of the declarations of our DSL. We generate a readable document that gave the engineer the possibility to validate and communicate his choice. Usually, this document is the main artefact used by safety engineers. Here, the document can be generated when the model is updated. We also generate code that eases interfacing with the output of the simulator and checking the violations. The code enables the user to analyze and identify rule consistencies.

4) Establishing satisfiability to avoid inconsistencies: using SAT Solver. SAT Solvers have been used in many practical applications. We expect to enable safety engineers to create a resilient and safe driver monitoring system that checks safety rules defined previously and investigates inconsistencies, possibly assigning priorities to sort them out. The SAT problem is a decision problem, which, given a propositional logic formula, determines whether there is an assignment of the variables that makes the formula true [15]. This will test rule inconsistencies, and verify solutions for all the rules. All logic operations for rules specification are translated to specific forms of coding. The task comprises of testing the rules with specific formulas by auto-generating specific checks for each rule.

B. User Process

The user has three tasks to analyze and guarantee safety in real or simulated scenarios, as seen in Fig. 4. First, he must describe the environment (scene, parameters, and properties), the behaviors (alarms and actions), and the security rules with priorities. Fig.5 is an example of rules (also called goals) introduced on the user interface. We define a ***RuleBasedPlanner*** named ***rbp***, referring to a scene defined in another file where we introduce the capacity of ego's perception using another syntax. Two properties (***prop1*** and ***prop2***) were previously defined having different states in Fig.6. Libraries of actions and alerts were also created. Conditions are put together through logical expressions.

Safety engineers assign every goal to a type that is either a priority or a constraint. If a goal has a priority on another, the second one should not be executed if the first one is true. If a goal has a goal type as a constraint, both are executed in parallel. This priority-constraint categorization helps the engineer choose which rule should be accomplished before or at the same time as another one. It implies a hierarchy of priorities between all the rules. In Fig.7, we show an example of two contradictory actions triggered at the same time. Goal 1 consists of having three conditions : (1) following the PV, (2) respecting speed threshold, and (3) respecting safety distance of 2 s. The first goal leads to a light acceleration, contrary to the second goal that generates emergency braking. It consists

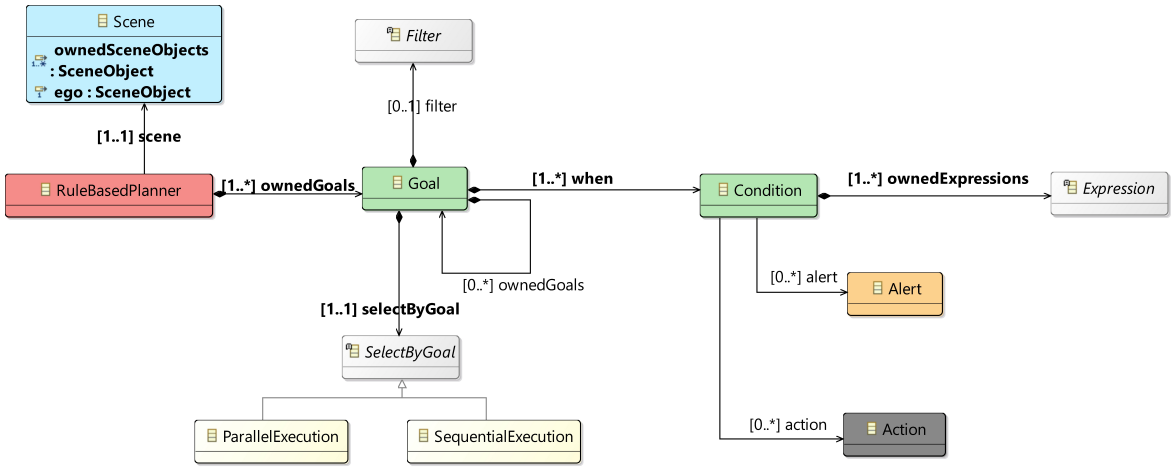


Fig. 2: RuleBasedPlanner Abstract metamodel

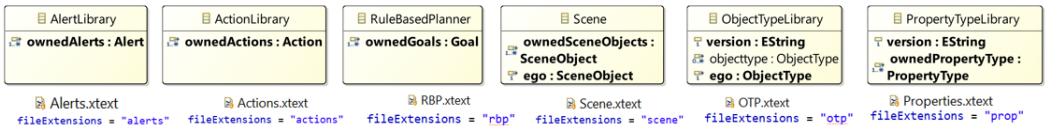


Fig. 3: Concrete Xtext Files in our metamodel

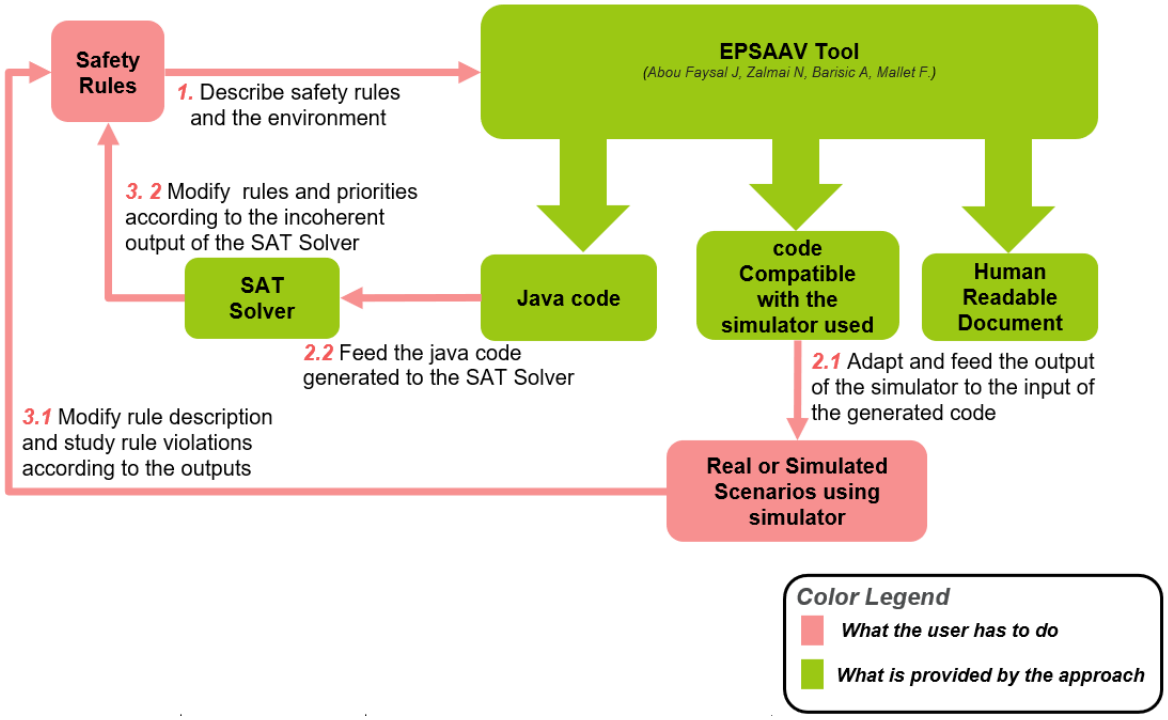


Fig. 4: User process to use the approach

of respecting a Time To Collision (TTC) for every Vehicle Road User (VRU), e.g. pedestrians. In this case, emergency braking has a strong priority over light acceleration, and this priority needs to be carefully defined within the priority-constraint sorting.

When the environment definition is finalized, we execute

```
RuleBasedPlanner rbp{
    scene ^Scene
    GOAL goal1{
        GoalType Constraint
        WHEN {
            AND (
                propertyType "Properties.prop1" is "Properties.prop1.state1",
                OR (
                    propertyType "Properties.prop2" is "Properties.prop2.state2",
                    propertyType "Properties.prop2" is "Properties.prop2.state3"
                )
            )
            action  "Actions.action1"
            alert "Alerts.alert1"
        }
    }
}
```

Fig. 5: Formal Rules using logical expressions described by the user.

```
//this is the definition of the library for the Ego properties

PropertyTypeLibrary Properties
{
    version '1'
    state "prop1" CanBe
    "state1"
    "state2"
    state "prop2" CanBe
    "state1"
    "state2" operator">=" value 2.0 unit "s" //[operator] [float] [valuetypes e.g m/s]
    "state3" operator">=" value 1.2 unit "s"
}
```

Fig. 6: Properties and states using formal syntax defined by the user.

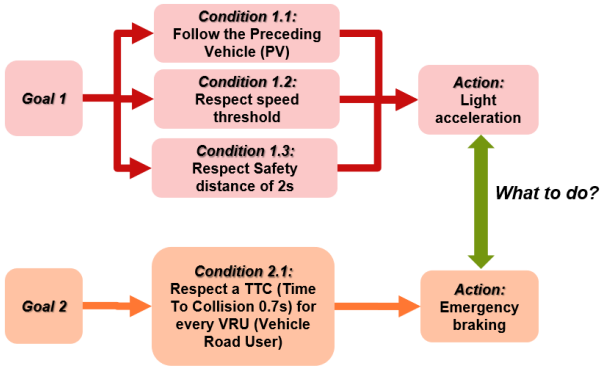


Fig. 7: Case where emergency braking should have a stronger priority over light acceleration.

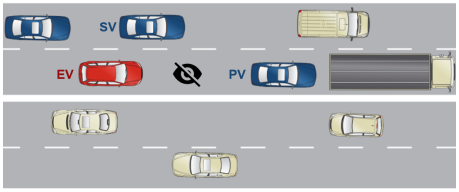
an automatic generation of monitors and a human-readable document. The main interest of this framework is to give the engineer the ability to define all the rules that he thinks should be triggered. It also gives him the ability to track rule modifications. In case of changes, the tool has the potential to generate a new code according to the specified rules. It is a generic tool for flexible rules that can be used not only for safety domains but also for security and failure domains. Depending on those rules, a specific code is then generated related to those formal rules. The document legibly describes the rules and libraries, so it helps to communicate the accuracy and completeness of those rules. The monitors allow the user to adapt perception data with safety rules and investigate rule inconsistencies. For the generated Java code, we feed it to the SAT Solver to test inconsistencies and modify safety rules. For the code compatible with the simulator used, the user has to take the generated monitors and plug them into real or simulated scenarios such as Webots [16], Carla [17]...
Then, an adaptation of our input data with the output data of the simulator is necessary to test, verify and identify rule violations using formal languages. This part is further detailed in our Renault use case in section II-C where we detect ambiguities in rules that may lead to violations.

C. Renault Use Case

To apply operational safety on the trajectories of self-driving cars, we will need the elementary data necessary for the system, such as rules and libraries. We translate scenarios, their risks and the measures to be taken, to formal rules describing lateral and longitudinal control. The EPSAAV tool facilitates this formalization process and improves communication between the engineers. We proceed to an auto-generation of a human-readable document and monitors, that will be helpful to acheive a SAVI.

1) Formal Goals Description: we introduced five scenarios in our use case as shown in Fig.10 that deal with risks of no/insufficient or unexpected braking, no lateral correction, and unexpected lane change. We take a scenario that describes the case where the Ego vehicle (EV) is having a problem detecting lines and/or Preceding Vehicle (PV) in Fig.8a. The risk we have is a lateral collision with the Side Vehicle, or Straddling Vehicle (SC). In case of a line or PV loss, the system shall maintain the EV in the lane using the remaining information. If the line disappears more than t6 seconds, the system must trigger an ***Emergency Operation*** (***EOP1***). Fig.8b formalizes this scenario by creating a goal that contains one condition to avoid the risks and to trigger behaviors. Goals can have multiple conditions. For each condition (*WHEN*), there is logical expressivity that constitutes the syntax.

2) Generation of documents and monitors: using the EPSAAV tool, we generate a readable document that gave the engineer the possibility to validate and communicate his choice. We also generate (a) C code which eases interfacing with the output of the simulator and checks the violations, and (b) Java code which enables the user to analyze and study rules incoherences. The C code is compatible with Renault's simulator called *"FusionRunner"*. We chose *"FusionRunner"* among all simulators for many reasons: (1) it executes perception's algorithm and sensors fusion data of Renault, (2) it runs driving data on open roads and many other real-life or simulated scenarios, and (3) it provides practical information (such as data sensor, data fusion, TTC, PT, Autonomous Emergency Braking (AEB), Adaptive Cruise Control (ACC), ...). We integrate this code into the perception algorithm and then adapt the safety functions by feeding the simulator's output to the input of the generated code.
The auto-generation consists of: (1) translating all the environment defined to functions compatible with Renault's language, (2) filling the functions needed to test rules according to defined thresholds, properties, and parameters



(a) Lateral control scenario: no lane/PV detection case.

```
when
    S_stable_control is stable AND
    (
        S_front_car_tracking is disappeared_more_than_t1 OR
        S_line_detection is no_detection_in_more_than_t6 OR
    )
then goal:
    executing A_emergency_operation_1
```

(b) rule specification formalized for this use-case scenario.

Fig. 8: Formalizing scenario to goal containing one condition and using logical expressiveness in case of a no lane/PV detection scenario.

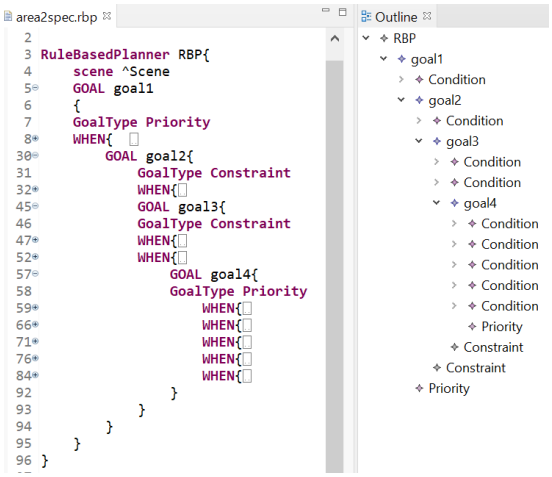


Fig. 9: Five formal rules introduced in our Renault Use Case with parallel and sequential executions.

Longitudinal Control	*Risk of no/insufficient braking*	Deceleration
	Risk of unexpected braking	False recognition
Lateral Control	*Risk of no lateral correction*	No lane detection
	Risk of unexpected lane change	Wrong lane detection
Lateral & Longitudinal	*Risk of no/insufficient braking /no lateral correction*	Swerving

Fig. 10: Five scenarios violations introduced in Renault's use case.

given in the beginning, and (3) creating a function that treats all goals and conditions in priority or parallel, depending on the goal type. We took the C auto-generated code and implemented it into the *"FusionRunner"*.

3) Interfacing the auto-generated C code with Renault's simulator: Fig.11, Fig.12, and Fig.13 constitute the windows output to better visualize violations defined in the rule-based planner. In Fig.11, we can visualize a real-time video in the Fusion Context View and the sketch of this video in the Fusion Display. In Fig.12, and Fig.13, we can see all the binary states' properties, five goals with their conditions, actions, and alerts triggered in the SAFETYCHECKER window output. We store, at each slot, the parameters' values from fusion data. The windows output helps us inspect

Fig. 11: Fusion Control View and Fusion Display windows at step 4217.

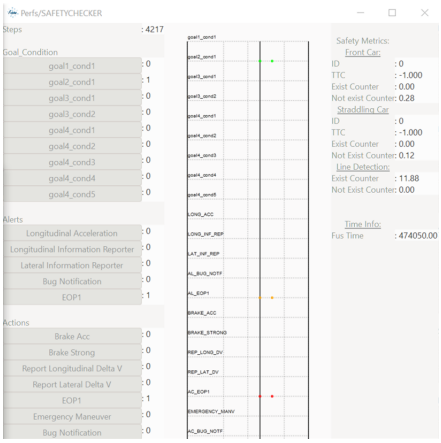


Fig. 12: SAFETYCHECKER Window for safety analysis of violations that shows the parameters, the goals with the alerts and actions triggered at step 4217.

violations and ambiguities in the goals declaration.
The results in the following section indicate that it is possible to reuse the model defined to verify safety in the automotive industry. It also shows the benefits and efficiency of using our DSML. In addition, it makes safety exploration easier for engineers, therefore improving the quality of their surveys.

4) Safety Analysis of Violations and Inconsistencies (SAVI): an ambiguity is presented in a violation that occurred in Fig.12

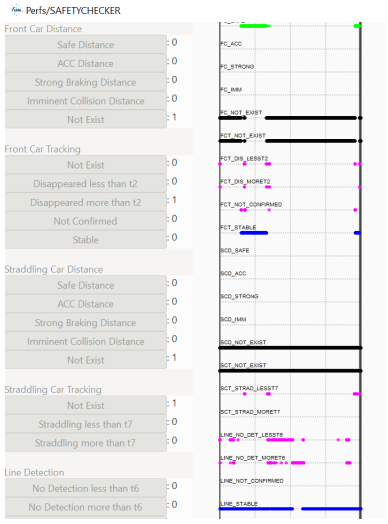


Fig. 13: SAFETYCHECKER Window for safety analysis of violations that shows the properties binary states at step 4217.

at step 4217. This violation is assigned to the *goal2_cond1* presented in 8b. The ***EOP1*** is triggered according to this goal when the PV disappeared more than a threshold. This is ambiguity because if we look at the Context View in Fig.11 at that step, we can see that there is no PV, and it is not logical to trigger an ***EOP1***.

We propose a modification for the safety rule to erase this ambiguity by adding a condition on the stability for the line detection as shown in Fig.14. To study rule inconsistencies,

```
when
    S_stable_control is stable AND
    (
    NOT ( S_line_detection is stable) AND
        (
            // 2.4.1.1
            S_front_car_tracking is disappeared_more_than_t1 OR
            // 2.4.1.1, 2.4.1.3.1
            S_line_detection is no_detection_in_more_than_
        )
    )
then goal:
    executing A_emergency_operation_1
```

Fig. 14: Modification in the rule expression to erase ambiguity.

the generated Java code fed to the SAT Solver will help us verify all solutions for all the conditions and goals. This work is still in process. By that, we will be achieving a SAVI.

III. Conclusion and Perspectives

This paper introduces a methodological proposal for using the MBSE approach in the automotive safety field. We describe the language development viewpoint where we talked about the abstract and concrete parts, the auto-generation of monitors and documents, and the SAT solver to study the inconsistency. We detail the user process tasks. The user has to specify requirements formally using EPSAAV to help him generate what he needs for safety analysis. We also show code generation that the user needs to link with the simulator's perception data. In our use case, we generated C code and tested and visualized goals to demonstrate that this approach is feasible. We show a goal's ambiguity, and the notifications triggered to the user. We propose a modification to delete an uncertain violation. If we could find all violation ambiguities and analyze inconsistencies, we can assure that all specifications are realizable and complete. We can apply this generic tool to test rules other than safety domains, such as security or failure domains. Ethic people can apply this framework to two different sets of rules. They can then select what is the best set of generated monitors. For future work, we will auto-generate java code for the SAT Solver. The SAT problems will help check rule inconsistencies and achieve SAVI analysis. We will also test rules on more real-life scenarios and analyze their output on more use-cases.

References

[1] M. A. Gosavi, B. B. Rhoades, and J. M. Conrad, "Application of functional safety in autonomous vehicles using ISO 26262 standard: A survey," in *SoutheastCon 2018*. IEEE, 2018, pp. 1–6.

[2] C. Ackermann, J. Bechtloff, and R. Isermann, "Collision avoidance with combined braking and steering," in *6th International Munich Chassis Symposium 2015*. Springer, 2015, pp. 199–213.

[3] Y. Sirgabsou, C. Baron, C. Bonnard, L. PAHUN, L. Grenier, and P. Esteban, "Investigating the use of a model-based approach to assess automotive embedded software safety," in *13th International Conference on Modeling, Optimization and Simulation (MOSIM20)*, AGADIR, Morocco, Nov. 2020. [Online]. Available: https://hal.laas.fr/hal-02942695

[4] J. Kapinski, J. V. Deshmukh, X. Jin, H. Ito, and K. Butts, "Simulation-based approaches for verification of embedded control systems: An overview of traditional and advanced modeling, testing, and verification techniques," *IEEE Control Systems Magazine*, vol. 36, no. 6, pp. 45–64, 2016.

[5] K. Falkner, V. Chiprianov, N. Falkner, C. Szabo, and G. Puddy, "A model-driven engineering method for dre defense systems performance analysis and prediction," in *Handbook of research on embedded systems design*. IGI Global, 2014, pp. 301–326.

[6] M. Bunting, Y. Zeleke, K. McKeever, and J. Sprinkle, "A safe autonomous vehicle trajectory domain specific modeling language for non-expert development," in *Proceedings of the International Workshop on Domain-Specific Modeling*, 2016, pp. 42–48.

[7] S. Shalev-Shwartz, S. Shammah, and A. Shashua, "On a formal model of safe and scalable self-driving cars," *arXiv preprint arXiv:1708.06374*, 2017.

[8] C. Tessier, C. Cariou, C. Debain, F. Chausse, R. Chapuis, and C. Rousset, "A real-time, multi-sensor architecture for fusion of delayed observations: application to vehicle localization," in *2006 IEEE Intelligent Transportation Systems Conference*. IEEE, 2006, pp. 1316–1321.

[9] M. Gao and M. Zhou, "Control strategy selection for autonomous vehicles in a dynamic environment," in *2005 IEEE International Conference on Systems, Man and Cybernetics*, vol. 2. IEEE, 2005, pp. 1651–1656.

[10] B. Schütt, T. Braun, S. Otten, and E. Sax, "Sceml: a graphical modeling framework for scenario-based testing of autonomous vehicles," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 2020, pp. 114–120.

[11] B. Combemale, O. Barais, and A. Wortmann, "Language engineering with the gemoc studio," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, 2017, pp. 189–191.

[12] B. Combemale, J. DeAntoni, M. V. Larsen, F. Mallet, O. Barais, B. Baudry, and R. B. France, "Reifying concurrency for executable metamodeling," in *Int. Conf. on Software Language Engineering, SLE*, ser. Lecture Notes in Computer Science, M. Erwig, R. F. Paige, and E. V. Wyk, Eds., vol. 8225. Springer, 2013, pp. 365–384.

[13] S. Kelly and J.-P. Tolvanen, *Domain-specific modeling: enabling full code generation*. John Wiley & Sons, 2008.

[14] J. Abou Faysal, N. Zalmai, A. Barisic, and F. Mallet, "Epsaav: An extensible platform for safety analysis of autonomous vehicles," *Advances in Model and Data Engineering in the Digitalization Era (MEDI 2021 Workshops)*, 2021.

[15] Borealis AI. Tutorial 9: Sat solvers i: Introduction and applications. Accessed: 2021-12-19. [Online]. Available: https://www.borealisai.com/en/blog/tutorial-9-sat-solvers-i-introduction-and-applications/

[16] Webots for automobiles. Webots user guide and reference manual. Accessed: 2020-06-05. [Online]. Available: https://cyberbotics.com/doc/automobile/introduction

[17] A. Dosovitskiy, G. Ros, F. Codevilla, A. M. López, and V. Koltun, "CARLA: an open urban driving simulator," *CoRR*, vol. abs/1711.03938, 2017. [Online]. Available: http://arxiv.org/abs/1711.03938