Formal rule-based scenarios for the design of safe autonomous vehicles

Joelle Abou Faysal

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Scénarios Formels Basés sur des Règles pour la Conception de Véhicules Autonomes Sûrs

Joelle ABOU FAYSAL

Renault Software Factory, Université Côte d'Azur, CNRS, Inria, I3S

Présentée en vue de l'obtention du grade de docteur en informatique à l'Université Côte d'Azur

Dirigée par : Frédéric MALLET
Co-encadrée par : Nour ZALMAI
Ankica BARISIC
Soutenue le : 9 Juin 2022

Devant le jury, composé de :
Rapporteuses :
Daniela CANCILA, Chargée de recherche, Commissariat à l'énergie atomique et aux énergies alternatives (CEA)

Amel MAMMAR, Professeur, Telecom SudParis

Renault Group Software Factory cnrs i3s sophia antipolis Inria

Acknowledgements

First of all, I would like to express my gratitude to the Université Côte d'Azur establishment, for giving me the chance to complete my studies. I thank each of the committee members of this study, Pr. Amel Mammar and Dr. Daniela Cancila, for their time and their generous and thoughtful contributions. I would also like to thank them for making my defense a pleasant moment, as well as for their brilliant comments and suggestions. I would like to express my gratitude and my special thanks to my supervisors Dr. Ankica Barisic, Dr. Nour Zalmai and Pr. Frédéric Mallet, they have been great mentors for me. I would like to thank them for encouraging my research and allowing me to grow as a research engineer. Their advice both on research and on my career has been invaluable. Ankica, Thank you for supporting me for everything. You are such an amazing person who provided me positive energy and immense knowledge. I have been extremely lucky to have a supervisor who cared so much about my work, and who responded to my questions and queries so patiently and promptly. Nour, thank you for accepting to be my advisor and for guiding and supporting me over these past two years. This fruitful work would not be possible without you. I want to thank also Jean-Pierre Giacalone who was my first supervisor at Renault and gave me the chance to start this thesis.

I owe a huge debt of gratitude to Kairos team directed by Pr. Robert De Simone who welcomed me and provided feedback and encouragement. I also thank Dr. Julien DeAntoni for helping me and organizing presentation support at crucial times.

I thank my team at Renault directed by Guillaume Menez for welcoming me into the team. I got an incredible Ph.D. during these three years. I want to thank Cedric Leclerc, Francois Amand, and Benoit Sainson for allowing me to work in good environmental conditions. I want to thank also Nadege Rizet and Frederic Ronconi for their time and for giving me the safety support needed for my thesis. I am lucky to get a job opportunity with Renault in the Dea-LA team directed by Dr. Laurent Pahun and Marc Peresse. I am glad that my first working experience will start with you.

A special thanks to my family. Words can not express how grateful I am to my mother, and father for all of the sacrifices that you have made on my behalf. I would also like to thank my beloved brother Jean and sister Joya. Thank you

for supporting me in everything, and especially I can't thank you enough for encouraging me throughout this experience.

I thank my best friends Lama, Sally, Ghadir, Maroua, Farah, and Layal for listening, guiding, encouraging, and advising, with wisdom and good humor always. Thank you for the valuable and timely feedback at key stages in the research process and the personal motivation.

Thanks also to my wonderful best friends and colleagues in Renault, Sarra, Tarek, Berkay, Mathieu, Ugo and Raj for daily sustenance in so many ways. A special note of thanks to Dr. Youssef Bou Issa who is one of the reasons I am here today, and to Dr. Kabalan Chaccour whom both provided feedback and encouragement.

Abstract of the thesis

Title: Formal rule-based scenarios for the design of safe autonomous vehicles

The automotive industry is preparing to deploy fleets of Autonomous Vehicles (AVs), which raises both technical and social challenges. Indeed, acceptance of AVs from both users and public authorities requires proof of the safety, reliability, and security of the proposed solutions. Major errors from AVs cannot be tolerated by society who expects a close-to-zero accident policy. Considering the wide uncertainty in the AV's environment, we cannot use the otherwise successful, systematic, and exhaustive verification techniques. Therefore, we need to rely on the assume-guarantee approach in which we can enforce some guarantees on the vehicle's safety under a set of assumptions on the environment. This work addresses this topic by considering a scenario-based approach for known unsafe scenarios. Here safety is restricted to the acceptance of the Safety Of The Intended Functionality (SOTIF) standard, recently defined by ISO/PAS 21448.

One proposal to assess safety is to test AVs in real traffic and observe their behavior. As logical as it may sound, it is neither feasible nor sufficient because it poses significant risks to the environment and requires a lot of testing, which makes it non-scalable and that does not guarantee to cover 100% of the possible, however unlikely, scenarios. What we propose is to precisely define scenarios in which we describe the environmental conditions, and we provide the risks and the measures to be taken when a hazard is identified.

In this thesis, we design a process based on formal modeling and verification to give sufficient guarantees and build concrete evidence to ensure the safety of AV. We propose a formal language, called Extensible Platform for Safety Analysis of AVs (EPSAAV), to bridge the gap between safety requirements, usually expressed in a natural language, and the actual code embedded inside the vehicle. We use EPSAAV in the fashion of Model-Based System Engineering (MBSE) in a model-centric approach where artefacts and analyses are automatically derived from the model. We propose three instantiations of this language. First, we generate a safety report that can be used as a reference at the corporate level to disseminate the safety policy to be enforced in the company. Secondly, we generate a dedicated monitor to be interfaced with a chosen simulator. The operational semantics of this monitor are exploited in simulation to debug and identify ambiguities in scenarios. Typically this can be used by other teams to validate their

work with regard to the global safety policy of the corporation. Third, we rely on the formal semantics of this language to detect inconsistencies that may arise from the safety requirements. For this purpose, we build a dedicated verification engine.
Finally, to illustrate the feasibility of the proposed framework, we apply it to Renault-Nissan safety policy in full integration with the operational teams of Renault.

Keywords: Autonomous Vehicles, safety, formal rules, scenarios-based testing.

Résumé de la thèse

Title: Scénarios formels basés sur des règles pour la conception de véhicules autonomes sûrs

Les constructeurs automobiles se préparent à déployer des flottes de Véhicules Autonomes (VAs). Cela soulève des défis à la fois techniques et sociaux. En effet, l'acceptation tant par les usagers que par les pouvoirs publics nécessite de convaincre de la sûreté, de la fiabilité, et de la sécurité des solutions proposées. Les erreurs majeures commises par les AV ne peuvent être tolérées par la société, qui attend une politique proche du zéro accident. Compte tenu de la grande incertitude dans l'environnement des VAs, nous ne pouvons pas nous contenter d'utiliser uniquement les techniques de vérification systématiques et exhaustives. Par conséquent, nous devons nous appuyer sur l'approche hypothèse-garantie dans laquelle nous pouvons appliquer certaines garanties sur la sûreté de fonctionnement du véhicule à partir d'un ensemble d'hypothèses sur l'environnement. Ce travail aborde ce sujet en considérant une approche basée sur des scénarios pour les cas dangereux connus. La sûreté de fonctionnement se limite à l'acceptation de la norme Safety Of The Intended Functionality (SOTIF), récemment définie par la norme ISO/PAS 21448.

Une proposition pour évaluer la sûreté de fonctionnement consiste à tester des VAs en circulation réelle et à observer leur comportement. Aussi logique que cela puisse paraître, cette méthode est difficilement réalisable et largement insuffisante car elle présente des risques importants pour l'environnement et nécessite de nombreux tests. Cette méthode est donc non évolutive et non exhaustive car il n'est pas possible de couvrir tous les scénarios possibles, et notamment ceux très peu probables, avec une occurence très faible. Ce que nous proposons, c'est de définir précisément des scénarios dans lesquels nous décrivons les conditions de l'environnement ainsi que les risques et les mesures à prendre lorsqu'un danger est identifié.

Dans cette thèse, nous concevons un processus basé sur la modélisation formelle et la vérification pour donner des garanties suffisantes et construire des preuves concrètes pour assurer la sûreté de fonctionnement du VA. Nous proposons un langage formel, appelé Extensible Platform for Safety Analysis of AVs (EPSAAV), pour combler l'écart entre les exigences de sûreté de fonctionnement,

généralement exprimées en langage naturel, et le code réel intégré à l'intérieur du véhicule. Nous utilisons EPSAAV comme Model-Based System Engineering (MBSE) dans une approche centrée sur des modèles où les artefacts et les analyses sont automatiquement dérivés du modèle. Nous proposons trois instanciations de ce langage. Dans un premier temps, nous générons un rapport de sûreté de fonctionnement qui peut servir de référence au niveau de l'entreprise pour diffuser la politique de sûreté de fonctionnement à appliquer dans l'entreprise. Dans un second temps, nous générons un moniteur dédié pour s'interfacer avec un simulateur choisi. La sémantique opérationnelle de ce moniteur est exploitée en simulation pour déboguer et identifier les ambiguïtés dans les scénarios. Typiquement, cela peut être utilisé par d'autres équipes pour valider leur travail par rapport à la politique de sûreté de fonctionnement globale de l'entreprise. Troisièmement, nous nous appuyons sur la sémantique formelle de ce langage pour détecter les incohérences pouvant découler des exigences de sûreté de fonctionnement. À cette fin, nous construisons un moteur de vérification dédié.
Enfin, pour illustrer la faisabilité de l'approche proposée, nous l'appliquons à la politique de sûreté de fonctionnement Renault-Nissan pleinement intégrée aux équipes opérationnelles du groupe Renault.

Mots clés: Véhicules Autonomes, sûreté de fonctionnement, règles formelles, tests basés sur des scénarios.

Formal Rule-Based Scenarios for the Design of Safe Autonomous Vehicles

Joelle ABOU FAYSAL

Renault Software Factory, Université Côte d'Azur, CNRS, Inria, I3S

Presented for obtaining the degree of doctor in computer science at the University Côte d'Azur

Directed by: Frédéric MALLET
Co-supervised by: Nour ZALMAI
Ankica BARISIC
Defended on : 9 June 2022

In front of the jury, composed of:

Daniela CANCILA, In Charge Of Research, Commissariat à l'énergie atomique et aux énergies alternatives (CEA)

Amel MAMMAR, Professor, Telecom SudParis

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context and motivation

Nowadays, 90% of car accidents are caused by human errors, such as poor driving skills or indigent judgment due to a failure of human perception or the driver's lack of attention [6]. Moreover, errors made by road users, such as violations of traffic rules, affect traffic safety as well [7]. Sometimes a brake failure caused by a vehicle's malfunction, or even a lack of safety infrastructure induces loss of lives. To improve road traffic safety and security, automotive industries have started to invest a lot of money in deploying self-driving systems for transportation. France's road and traffic regulations have evolved to allow tests in controlled environments [8]. Replacing human drivers with (partially) automated systems aims at taking better decisions and putting efforts to reduce errors. The road is long as there are several intermediate levels of autonomy before reaching fully Autonomous Vehicles (AVs).

The main difficult hurdle in the autonomous domain is to guarantee that systems and software components are safe and secure. With the advent of AVs, engineers have witnessed several deaths and accidents that raise troubling questions about the safety and security of such vehicles and the current limitations of the technology. The question that arises is: who is responsible for a crash? It is important to show that AVs make better decisions than human drivers under all circumstances. It is also necessary to overcome these challenges before having big fleets of cars on the roads. To avoid rejection from public opinion, we need to get them involved in the adoption of good decisions.

It is, therefore, crucial to provide concrete evidence to be clear on the actual advantages of using AVs and be clear on what rules they should follow and what guarantees they offer. Safety is associated with guarantees that some conditions must be met when contingencies arise. If not, the behavior must be adjusted accordingly. In dynamic environments, real-time safety checks of the planned trajectories and the driver are sometimes lacking to ensure that the trajectories

obtained from trip planning are safe.

One proposal to assess operational safety is to test-drive AVs in real traffic and observe their behavior. As logical as this may sound, it is not efficient because it poses significant risks to the environment. We can look at the Uber accident [9] where the driver was doing mileage tests in Arizona. Several things went wrong while the driver was not paying attention to the road: there was no real-time driver safety check, and Uber did not have the resources in the vehicle to assess the driver's attention. Mileage testing requires a lot of time and testing to ensure safety, and there is no practical way to assess the coverage rate or to pinpoint rare, even though possible, cornercases [10]. Using a statistical approach to estimate "How many miles autonomous vehicles would need to be driven to demonstrate safety" leads to the assessment that 8 billion miles in 400 years with a fleet of 100 vehicles driving all the time would be necessary [10]. This is somehow unachievable leading to the conclusion that AVs cannot express safety capabilities based on road tests only [11, 12]. Simulation testing is certainly a viable alternative [13] that is only valid if we assess the external conditions under which we guarantee safe behavior. Consequently, the tests are conducted over limited Operational Design Domains (ODDs) under which one can define precisely both the assumptions and the expected safety requirements. However, such approaches offer no global guarantees with regards to the wide variety of situations, user behaviors, or driving conditions left unexplored. Even under limited ODDs, we still need rigorous approaches and formal language to specify the safety rules and enforce them in the context of real or simulated scenarios.

Such languages dedicated to the specific, very specialized community (here the safety experts) are known as Domain-Specific Languages (DSL) [14]. As they are specific, they demand the build ad-hoc specific support tools, even though more and more modeling framework strives to provide generic support and help reduce the development time of such tools, therefore increasing productivity [15, 16]. For the Safety domain, DSLs can be a solution to fill the gap between safety experts and embedded code developers. Bridging this gap is expected to increase safety engineers' productivity, validate the correctness of the requirements, and improve solutions' reliability. By safety engineer, we refer to the safety system engineer who has the task of defining software requirements related to functional and safety. Sometimes, engineers can experience practical difficulties when adopting DSLs [17]. The measure of success has to be determined by assessing the impact of correct specification by formally defining requirements and having a code generation that can help detect rules' violations and inconsistencies in a realistic context of use. In the typical software development lifecycle, we try to translate needs into natural language requirements and produce code that meets them. However, safety studies of operation are often carried out by specialists in the field, after the functional and organic architectures have been defined. This results in late detection of operational safety issues and loss of time due to the necessary iterations between the system architecture and the safety special-

ist engineer. This is why safety concerns need to be addressed from the early stages of the DSL life cycle so that practitioners can perform incremental safety evaluations.

Model-Based System Engineering (MBSE) is meant to use DSLs in a holistic fashion to provide a co-design with regular, short iterations. MBSE maintains global systems models, instead of raw text documents, as global shared modeling artefacts that should be used as a communication medium between engineers. It advocates for a model-centric approach where shared models are at the center of the process and different concrete artefacts are automatically derived from the domain model. Many approaches are following and applying MBSE to different domains, including safety and security verification [18]. Our work adheres to this principle and contributes to it in the context of AVs. MBSE helps to deal with the increasing size and complexity of systems [19] and allows to reason on the model before deployment. Formal modeling and verification in automotive systems are essential to provide sufficient guarantees, especially in the case of dangerous and unforeseen situations. Building a systematic safety evaluation approach is supposed to mitigate the risk of producing inappropriate solutions that often cannot be reused. This work is expected to enhance the community awareness of the relevance of Domain-Specific Modeling Languages (DSML) for safety assessments to bridge the gap between the safety engineers and code artefacts provided by embedded code engineers.

1.2 Research problems and questions

The explosion in the complexity of vehicle safety assessment, therefore, imposes methods of high-performance and robust developments in vehicle safety. Operational safety defined in the ISO 26262 standard [20] implies having a design for the safety component that deals with all unsafe situations. This standard requires several demonstrations of compliance with the rules in the development of complex and critical functions. So far, the engineering activities in the system design process and the safety were separate and sequential. For example, the system architecture defines system performance, and the safety specialist is interested in the system when it malfunctions. He must therefore enrich the functional analysis of the system with the dysfunctional aspects, and return results after a certain delay. The conclusions of the analysis are therefore partially valid. This misalignment also results in producing and testing code in which they discover errors between what was intended and what was built. The same problems occur when testing scenarios. This is where the Safety Of the Intended Functionality (SOTIF) standard came to help the engineer not only to test requirements but to perform scenario-based testing to deal with more complex situations treating higher levels of autonomy. Our research work tackled the following problems:

1. **Lack of a systematic approach for scenario-based safety evalua-**

tion of AVs environment. Most of the existing evaluations use traditional manual approaches to enforce safety decisions [21]. There is a threat to using these approaches, as safety experts' analyses depend on their experiences. Sometimes safety rules are not well formalized and usually are not reusable in subsequent development. In addition, classical exhaustive verification techniques cannot guarantee that the system is safe because of the high degree of uncertainty in the environment. Safety experts need to provide rapid production of a growing number of complex software tasks. Their degree of specialization is pushing for the involvement of domain concepts in the software development process. There is a necessity to assess the impact of introducing a DSL in their domain and evaluate it. There is also a need for a tool that can deal with situations when the environment gets more complex. Safety experts define requirement-based tests as focusing on malfunctions rather than environmental conditions. This is why presenting a system that helps the safety engineer formally introduce his rules considering the outside conditions is crucial.

2. **Lack of operational solutions to support the analysis of safety specifications.** Safety experts usually produce requirements using natural language. Therefore there is a big gap between what they express and the code artefacts or variables that implement the solution. Reducing this gap to provide analysis support at the level of safety specifications is of key importance. Domain experts give these specifications to the developers who integrate them into the simulators. A gap remains between safety and the implementation of the system which begins to widen as the rules become more complex. Automation is needed to fill this gap, which gives the safety expert more time to spend on rule specification.

3. **Inability to detect violations in real or simulated scenarios while detecting errors and ambiguities of defined requirements.** In order to assess all safety experts' notifications, it is necessary to provide an approach that shows the implication of the defined requirements on real scenarios. It helps better visualize exploited safety defined specifications by producing to the interpreter engineer warnings and notices during an infringement. This improves testing rules and trigger alerts and actions to the user to help him avoid collisions when transgressions occur. For instance, 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. The safety expert may have missed a specific requirement making his performed evaluation study not mature enough. This is why it is primordial to detect the violations' origins and visualize the provenance of the ambiguity in its defined requirement.

4. **Inability to automatically detect inconsistencies from the defined requirements and specifications.** In order to evaluate all safety experts' documents, it is necessary to provide an approach supported by adequate tools which can be integrated into safety rules. For instance, the vehicle must come to a complete stop if a pedestrian is near the crossing line. At the same time, vehicles should maintain a specific speed to reach a comfortable driving level for the passengers. Most importantly, the AV should be aware that pedestrians can be obstructed by a traffic sign and may step out in front of you. The highest priority is placed on entities with an associated risk of collision which in this case is for the pedestrian. Sometimes, the safety expert does not pay attention to the parallel or priority executions between the defined rules. This is why it is radical to uncover requirements inconsistencies and systematize rules' order. Presenting an automation system that helps the safety engineer automatically detect inconsistencies in their defined requirements is essential to speed up their analyses.

The deployment of safety is interesting from Renault's point of view. With many organizations developing their safety assessment languages or hiring companies to develop such languages for them, our framework can help them perform safety evaluations on scenarios and reach more of the missing requirements and specifications. The main challenges while addressing the problems identified in this dissertation were:

1. **Defining an appropriate level of abstraction for Safety domain considering AV's environment.** Most of the existing safety evaluations are performed ad-hoc, not reporting enough details of the experimental design or analysis result. This research work produced a general safety model for AV in Chapter 3, tailored for conditional environment and rules evaluation, and instantiates it in a case study for Renault in Chapter 6. Integrating the safety assessment process for scenario-based testing with the DSML development was an additional challenge. Both the Functional Safety domain evaluation and DSML development process are complex and evolving. Therefore, we discussed the safety criteria and proposed a development and evaluation process that can be used to achieve formalization to inject safety rules (Sections 3.3.2 and 3.3.3). Using the MBSE approach to assess software safety, enables formalization, improves reuse of the software, and helps to address safety analysis [22].

2. **Developing the tool to support the analysis of safety specifications.** It was challenging to capture the complexity of information and process to generate code and human-readable documents. It was also arduous leveraging it systematically. We developed a tool that supports discovered knowledge, validates assumptions, and provides an automation scheme. This knowledge is presented in a formal model which captures only the

meaningful information, helps with traceability and development decisions, and produces desired monitors and documents (Section 3.3.5).

3. **Applying safety assessments to DSML development in an industrial context to visualize violations and detect ambiguities of defined requirements.** The DSMLs are developed for different domains, each of them having users with different background knowledge and the necessity to understand specific concepts. It was challenging to apply our approach in the development of DSMLs for automotive safety contexts. However, we succeeded to apply our approach to real and simulated scenarios by connecting our DSML with the simulator used in Renault which can guarantee all unsafe cases (Chapter 4). The user is alerted with specific information and assumptions that might be linked to the motion planner shortly.

4. **Applying safety assessments to DSML development in an industrial context to verify inconsistencies in safety engineers' choices.** By studying rule inconsistencies, the engineer is informed if there is a specification misplaced in its rule-based planner or if it should be triggered in priority. We succeeded to apply our approach by connecting our DSML with a Solver that helps study logic expressiveness on rules. This aids the expert to modify his alternatives by adding or removing priorities on rules (Chapter 5).

The objective of the research is to promote the environment of the AV in the use of DSMLs by building up a conceptual framework that supports their development process, leveraging safety as a first-class concern. This involves the integration and adaptation of the current evaluation methodologies, their concepts, methods, tools, processes, and metrics. As briefly presented in Section 1.2, the aim of our research consists in providing contributions for the following major problems faced in the realm of DSML safety evaluation:

- **RP1**: Lack of a systematic approach for scenario-based safety evaluation of AVs environment.
- **RP2**: Lack of operational solutions to support the analysis of safety specifications.
- **RP3**: Inability to detect violations in real or simulated scenarios while catching errors and ambiguities of defined requirements.
- **RP4**: Inability to automatically detect inconsistencies from the defined requirements and specifications.

In particular, we address the following research questions:

- **RQ1**: How to model the safety evaluation for AVs environment?
- **RQ2**: How to support the analysis of safety specifications?
- **RQ3**: How to promote detection of violations in real or simulated scenarios while catching errors and ambiguities of defined requirements?
- **RQ4**: How to promote automation of inconsistencies detection from the defined requirements and specifications?

The **RQ1** is related to **RP1**, **RQ2** to **RP2**, **RQ3** to **RP3**, **RQ4** is related to **RP4**. Each of the above research questions is also related to the research hypotheses in the following Section.

1.3 Collaboration between Kairos and Renault Software Factory

Renault Group currently offers advanced driver assistance systems (ADAS) on its vehicles. These systems improve safety and provide convenience and comfort features. They serve as a gateway to AVs, even if they are initially designed to only assist the driver, who remains in control of the vehicle. Renault's goal for AVs is to change the car driving experience by reducing the risk of accidents and making commutes less stressful and more productive. Renault Group has launched pilot projects to study the behavior of autonomous Renault vehicles.
I had the opportunity to do the thesis in collaboration with Renault and Kairos, a joint team between Inria and I3S Laboratory. This thesis helps to support these studies by providing a language to inspect rule violations and inconsistencies. It helps safety experts improve, modify or even add their own safety goals. At Renault, I am part of the ADAS team managed by Guillaume Menez. My supervisor is Dr. Nour Zalmai. I also work in the Kairos research team led by Robert De Simone and my work has been supervised by Pr. Frederic Mallet with the assistance of Dr. Ankica Barisic.
Kairos works on methods and tools for temporal constraints at different levels of abstraction. More specifically, the team promotes the use of logical (formal) time from requirements to analysis and is interested in Model-Based Design approaches where analyses are conducted on models before generating various artefacts for simulation, monitoring or verification. These MBSE approaches are a solution to verify security and safety in the field of AVs. Kairos goal, however, is to benefit from all the industry-grade knowledge and experience embodied in these models and environments. This thesis exposes the need to complete and harmonize these models and their associated DSML representation, to achieve operational semantic continuity.

1.4 Thesis contributions and outline

1.4.1 Research contributions

The motivation of this work is to provide a systematic methodological approach using DSML to evaluate the functional safety requirements of AVs targeting safety experts. The design science methodology [23, 24, 25] fosters the creation of artefacts that are driven to problem-solving projects. Wieringa [26] regards design science projects as a set of nested regulative cycles that solve practical (i.e. engineering) and knowledge (i.e. research) problems that are decomposed into subproblems. The preceding pages explained the industrial context and the need to define a DSML for AVs. The hypothesis that is verified in this thesis is that it is possible to integrate the Safety domain and DSMLs to carry out safe operational systems.

- To answer **RQ1** when bringing integrating the Safety domain in the modeling language, it is necessary to check that we fuse the mandatory elements and share vital terminology. For this, the first contribution of this thesis is to propose a conceptual model using DSML for defining AVs environment to evaluate the safety, which we called Extensible Platform for Safety Analysis of AVs (EPSAAV) presented in Chapter 3. The use of DSML enables fast prototyping of AV's environment by using a metamodeling structure [27]. Endowed with formal semantics it brings a possibility to assess AV's environment before generating a conforming monitor. Therefore it makes it possible to define the safety goals, corresponding to a deliverable required by the SOTIF standard.

- The second contribution of this thesis that answers **RQ2** is a methodology implemented in the DSML that supports the analysis of safety requirements and validates safety experts' choices and helps them in the development process (Chapter 3).

- The question then arises whether the safety goals can lead to violations and be used to detect ambiguities from the defined requirements. Our third contribution is complementary to the second one and responds to **RQ3** (presented in Chapter 4). The safety of the intended functionality, where intended means specifications, another deliverable required by ISO/PAS 21448, is an extracted view of this process. This contribution leads to determining Safety Analysis of Violations and Inconsistencies (SAVI) as an iterative process to modify the defined requirements described in Chapter 6.

- The last question addressed here is whether or not the safety goals are consistent and actually express the intention of safety engineers. Our fourth

contribution is complementary to the second one and responds to **RQ4** (presented in Chapter 5). We study rules inconsistencies to verify sequential and parallel rules. This contribution leads also to a Safety SAVI process by modifying the defined requirements (Chapter 6).

1.4.2 Outline

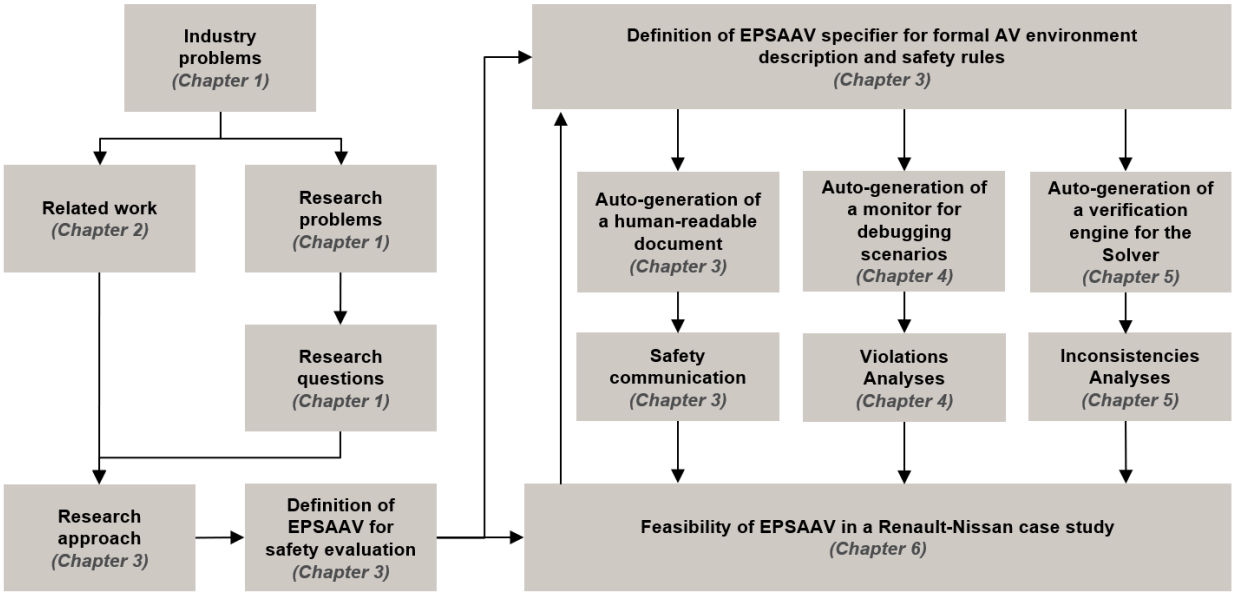

Figure 1.1: Research process overview.

This thesis is divided into the following major parts as presented in Figure 1.1:

- context and related work that detail the problem definition in Chapter 2. We introduce the reader to the context of safety. It is followed by a description of the methods and standards for safety specifications where we analyze the related approaches. We also introduce the context of Model-Driven Engineering where we present Domain-Specific Modeling Languages (DSMLs) and their life cycle. This Chapter also highlights the scope of related works, their benefits, and shortcomings for solving the problem addressed in this thesis.

- proposal of a systematic approach

  – Chapter 3 introduces concepts and technologies that are crucial for the argumentation of our proposed solution using DSML. It presents the Extensible Platform for Safety of AVs (EPSAAV) conceptual framework as a proposed solution to the given problem. This takes into

account AV's environment and also follows up with the experimental model for DSML and patterns for usability. The proposed DSML generates human-readable documents and monitors to be carried out in the following Chapters.

  - Chapter 4 describes how monitors are generated by the proposed tool EPSAAV and interfaced with the simulator used in Renault and called *FusionRunner*. Based on this adaptation engineers can analyze violations and detect ambiguities in the requirements.
  - Chapter 5 describes our verification engine based on a SAT Solver to detect inconsistencies in the rules themselves and help safety engineers build meaningful ones.

- applicability of proposed approach: Renault's use case

  - Chapter 6 shows the feasibility of our approach and introduces a case study inspired by Renault's internal process. This study illustrates how the SAVI process works.

Finally, Chapter 7 discusses possible future work and concludes the thesis.

Chapter 2

Background and State of the Art

2.1 Introduction

The Autonomous Driving (AD) domain in a wide range of components leads to a situation where safety needs become increasingly demanding and complex. There have been many accidents and fatalities by self-driving vehicles which caused a debate about the current limitations of the technology. Car manufacturers are therefore faced with a growing demand for safety assessments in the behavior of their vehicles. All critical systems, such as AVs, do not only depend on an array of electronics, sensors, and computer systems, they also require strong safety and functional requirements to ensure these systems work as intended and are built to mitigate risks. To show availability and service guarantee, computer systems inside modern automated vehicles must be highly responsive to the external environment. Several standards have been provided to automotive companies to develop safer systems. We mention about the benefits of Domain-Specific Modeling Language (DSML) to accompany the design process and help build better safety requirements.

In this Chapter we introduce the safety standards ISO26262 [28] for Functional Safety (FuSa), and ISO/PAS 21448 the Safety Of The Intended Functionality (SOTIF) [29] used for AVs. We then compare both standards and talk about how SOTIF can ensure and complement FuSa. We talk about the methods used in these standards that safety engineers apply to evaluate safety. We describe the Model-Driven Engineering (MDE) technology used in the thesis, and how we introduce the concept of a domain using the life-cycle of Domain-Specific Language (DSL). Our goal is indeed to build a DSML for safety analysis enabling Model Testing (MT).

2.2 Overview automotive safety approaches

Many approaches to evaluate safety are currently being used in the automotive industry. The most famous four are the following:

- The first one is *"The Miles Driven"* to show that systems can perform autonomously without failure on the roads for a very large number of kilometers (or miles). It is also called *"Data-Driven Safety"* to show that self-driving cars are somewhat better than human drivers. This approach is problematic because we need a huge amount of miles to be driven, 30 billion miles [12], to have enough statistical evidence.

- Another approach called *"Disengagements"*, is when the driver either takes control when the autonomous software requests him, or when he feels the need to intervene. In 2017, Waymo drove 563,000 kilometers in California and experienced 63 disengagements for a rate of roughly one disengagement every 9,000 kilometers [30]. The primary causes of the 63 Waymo disengagements in order of frequency, were unwanted vehicle maneuvers, perception discrepancies, hardware issues, software issues, behavior predictions, and finally, a single case of a reckless road user. It's clear that the core tasks of perception, prediction, and behavior planning remain challenging problems, and much work still needs to be done.

- A third approach is *"Simulation"* to build a virtual world, sometimes called *digital twins*, within which the AV's software is driven miles as a way to achieve the huge "Miles Driven" targets that would be needed to make a statistical claim on the safety of the AV's decision-making capabilities. The problem with this method is that validating that the simulator faithfully represents reality is as hard as validating the driving policy itself. In [12] they proved that even if a simulator has been shown to reflect reality for a driving policy, it is not guaranteed to reflect reality for another driving policy.

- Another approach of *"Scenario-based Verification"* is a combination of using *The Miles Driven* and *Simulation* approaches. It claims to assure an AV's ability to make safe driving decisions. The basic idea is that if only we could concentrate on all the possible driving scenarios that could exist in the entire world where the AV is exposed to several environmental conditions. This diversity can be covered via simulation or even on closed track testing or on-road testing, and as a result we can be confident that the AV will only ever make safe driving decisions [12].

The most realistic approach to assessing safety is applying scenario-based testing via simulation and real-data scenarios.

2.3 Safety standards for AVs

In this Section, we discuss existing safety frameworks for automotive and low-level autonomy feature development, which are often used in assessing hardware and software failures in AVs. Many institutes and laboratories are aiming to work for AV safety standards such as IEEE and the Underwriters Laboratories. In this thesis, we are interested particularly in two international standards: the Functional Safety (FuSa) approach described in ISO 26262 and the Safety Of The Intended Functionality (SOTIF) approach that elaborates on ISO 26262 and is defined in ISO/PAS 21448.

2.3.1 ISO26262 FuSa standard

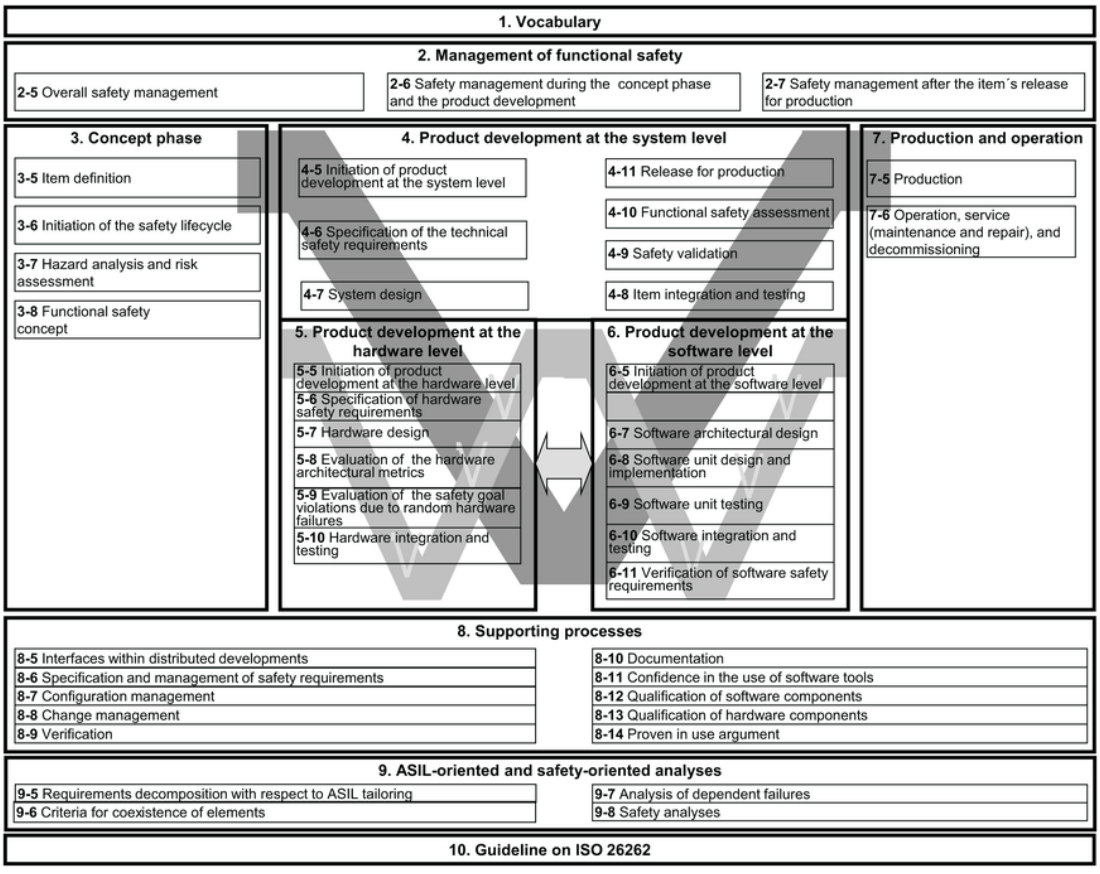

Figure 2.1: Overview of ISO 26262 in [1]

Let us start with the basic chests and concept of ISO 26262 Functional Safety (FuSa) of electrical and/or electronic systems [28]. FuSa is the absence of unreasonable risk from malfunctioning behavior caused by failures of hardware and software in a car, or unintended behaviors arising with respect to its intended

design. FuSa addresses the hazards that can affect AV safety. It does not address hazards related to electric shock, fire, smoke, heat, radiation, toxicity, flammability, reactivity, corrosion, the release of energy, and similar hazards, unless directly caused by malfunctioning behavior of E/E safety-related systems [1]. We give the example of a malfunction occurring during inadvertent braking. This serious case can injure or kill people in the car or around the car, and ISO 26262 cares about how to avoid this malfunction in an Electronic Component Unit (ECU) or even on communication between ECUs, and if these kinds of malfunctions occur, this standard proposes how to mitigate them to save people's lives.

The functional safety process is presented in an overall structure Figure 2.1. ISO 26262 is based upon a V-model as a reference process model for the different phases of product development. The shaded V represents the interconnection between the concept phase, the product development at the system, hardware, and software level phases, and the production and operation phase. We describe here the advantages of the ISO26262 standard.

- ISO26262 is a development method that improves previous workflows by testing components before deployment.
- It provides an automotive safety life-cycle. This life-cycle consists of the following management, development, production, operation, service, and decommissioning phases. It supports tailoring the necessary activities during these life-cycle phases.

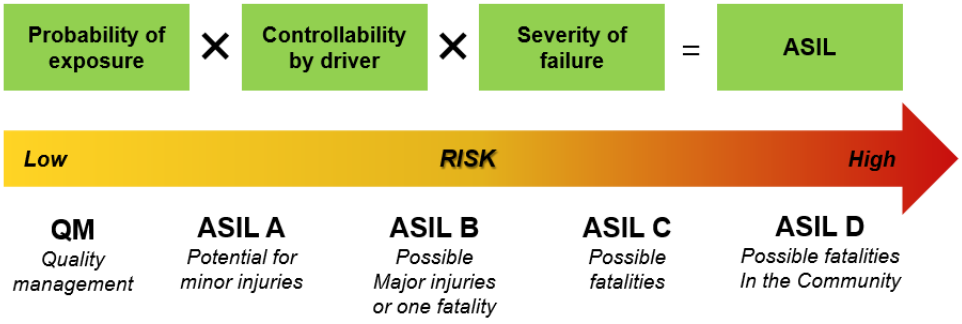

Figure 2.2: Automotive-specific risk-based approach called ASIL to determine integrity levels.

- ISO 26262 includes guidance to avoid these risks by providing appropriate requirements and processes.
- ISO26262 provides an automotive-specific risk-based approach to determine integrity levels, called ASIL presented in Figure 2.2. For example, a hazardous event combining S3 (fatal injury), E4 (high probability), and C3 (uncontrollable) is assigned to the ASIL D level. For many positions, there

is no ASIL level assigned, and Quality Management (QM) is listed instead. It means that there is no requirement to comply with ISO 26262 but the product must be developed in accordance with a QM process approved in an international standard such as NCAP [31]. It uses ASILs to specify applicable requirements of ISO 26262 to avoid unreasonable residual risk. ASIL replaces the concept of IEC 61508's SIL [32] which is based on the probability of failure per hour of use. In ISO 26262 the events that could cause injury are listed, and an ASIL is calculated (according to the table proposed in the standard). Three aspects are considered:

- Severity (S0 to S3): classifies the type of injuries that could result from this event;
- Probability of Exposure (E0 to E4): the probability that the event occurs in normal operation;
- Controllability (C0 to C3): classifies the difficulty of the driver to control the situation and avoid injury.

In this thesis, we focus on Part 6 which shows the ISO 26262 phase model for the product development at the software level in Figure 2.1. Part 6 deals with Software (SW) and includes tables of recommended (+) and highly recommended techniques (++) for different phases of the development process and each Automotive Safety Integrity Level (ASIL). The left side of the cycle covers the SW Requirements, SW Architecture, SW Detailed Design, and Code/Model Implementation. These phases are covered in the thesis by the proposed tool that follows the same cycle. EPSAAV language is a Domain-Specific Modeling Language that follows the DSL life-cycle presented in the following Section 2.8, for the safety domain presented in the ISO26262 life-cycle in Figure 2.1. On the right side of the cycle feature the software unit test and verification phases. Each of these phases aims to verify and validate if the corresponding design phase is implemented as required. The V model aims at maintaining the phases by tracing them.

2.3.2 ISO/PAS 21448 SOTIF standard

SOTIF has appeared as a new standard called ISO/PAS 21448 [29] to cope with the second edition of ISO 26262. SOTIF also talks about safety but it must be separated from functional safety. It is the absence of unreasonable risk due to hazards resulting from functional insufficiencies of the intended functionality or by reasonably foreseeable misuse by persons. We stop at SOTIF to talk about car faults and focus on the exterior of the car that can cause dangerous behaviors. One of the reasons for these dangerous cases can be functional deficiencies (such as bugs in the camera system resulting in no detection) and also foreseeable misuse by people (such as telling the driver to take over the wheel). The Operational

Design Domain (ODD) of the function consists of several use cases that contain trigger events related to external factors such as environmental conditions, road conditions, traffic conditions, or driver misuse. The hazards arising from these initiating events, when combined with specific operational scenarios, may lead to hazardous events that can result in damage.

Scenarios that may be encountered in the ODD of any automated driving function can be categorized as shown in Figure 2.3. In this thesis, we look at the

Figure 2.3: Evaluate safety for known/unknown safe/unsafe scenarios.

area 2 that is the known unsafe scenarios. The diagram in Figure 2.4 provides a graphical view of how all scenarios that may be encountered by an autonomous system in the field are categorized, also taken from [33]. At the start of SO-

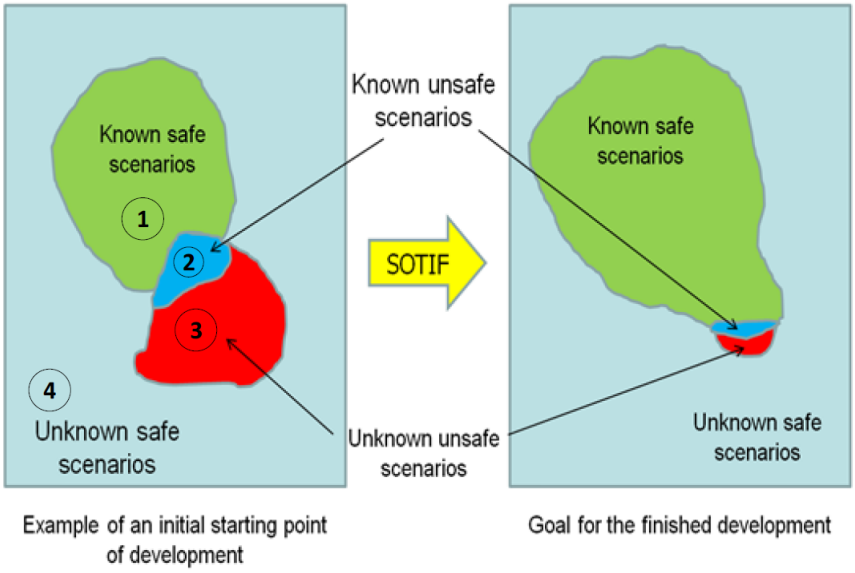

Figure 2.4: Evolution of scenario categories using SOTIF.

TIF activities, the area covered by the known or unknown dangerous scenarios is large, which leads to an unacceptable residual risk. The objective of SOTIF activities is to identify and reduce the number of hazardous scenarios such that the residual risk falls to an acceptable level. The objectives of SOTIF activities

are to maximize or maintain zone 1, minimizing zones 2 and 3. This maintains or improves the safe functionality. By identifying risks arising from known hazardous scenarios and implementing technical measures to improve function, we can minimize area 2. Once the measures have been evaluated through testing, the scenarios can be moved to area 1. By executing operations tests to validate and identify unsafe scenarios, unknown unsafe scenarios are minimized and moved to the second area. SOTIF activities provide an argument that the residual risk is acceptable. In this thesis, the main objective was to create a language to build and verify all requirements of known unsafe scenarios. Another advantage of SOTIF is that this standard presents an openness to new ideas by not providing details on how the process takes place. It gives high-level guidance to show the right direction but does not detail how exactly it should be done.

2.3.3 Combining FuSa and SOTIF to cover safety

Figure 2.5 shows what SOTIF can add to ISO 26262, and why we need more than one standard to cover all the risks and hazards related to automated functions. SOTIF requires scenario-based testing since it takes into account environmental

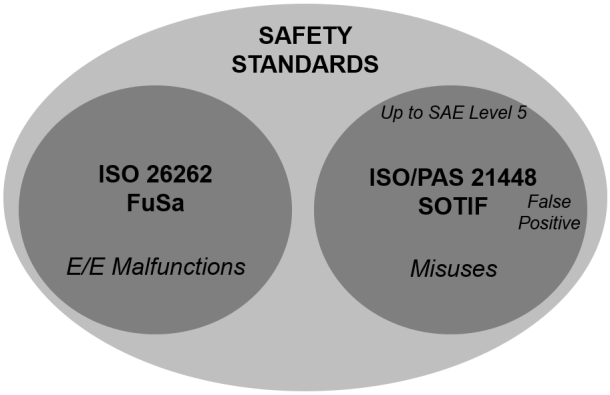

Figure 2.5: Overview automotive safety: FuSa and SOTIF.

conditions that can only be described in scenarios, unlike ISO 26262 which requires requirements-based testing since it describes failures that occur from the E/E systems and can be translated in form of requirements. Requirements-based testing was the main safety workhorse for a long time. It is a testing approach in which test cases, conditions and data are derived from requirements [34]. Figure 2.6 shows the typical design process for requirement-based testing workflow. It encompasses dividing requirements into small pieces creating an implementation model, and in parallel creating test cases that we call test vectors, and expected outputs. Test vectors and the expected output are designed in a requirement model in which we specify a reference model to early validate requirements. This implementation and expected outputs are used to verify the behavior of the actual

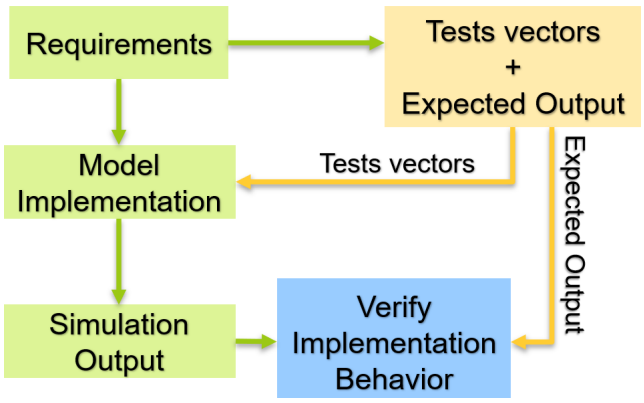

Figure 2.6: Traditional requirement-based testing workflow.

implementation. We can find many issues regarding this type of testing. First, it is difficult to measure the completeness of test cases against requirements. Second, it is very hard to tell if all test cases have covered all requirements in the implementation model, Finally, there is no way to verify all requirements independently of the implementation, so we are only showing how are we going to test unlike what to test. Now that we have identified that we can not put everything happening in the outside world only in the requirements, and the description of situations in the environment is not complete, then we can come to the end of this methodology to ask what is the value of the tests based on the requirements. Requirement-based testing comes to its limit. Scenario-based testing is a new methodology of testing to distinguish all situations and classify them. Automated driving needs to address the complexity of the outside environment and integrate it into the test cases for verification and validation to cover all scenarios and make them safe. This can work by classifying all critical parameters into different boxes to reduce testing efforts. The scenario-based testing approach comes to validation at the higher level to see that the final product has been achieved, and at the lower level, we have to use the requirements.

FuSa defines safety by creating item boxes around ECU sensors and alerting the system when faults occur inside to prevent cascading to the outside, unlike SOTIF which cares about the outside environment that leads to accidents or dangerous scenarios. So basically both standards study safety using goals to define requirements, but the techniques and methods are different. Comparing FuSa to SOTIF, FuSa has targets for certain hardware metrics, unlike SOTIF where the engineers has the freedom to choose the metrics they need to achieve a safe function and to provide an acceptable risk. The Society of Automotive Engineers (SAE) has defined six levels of automation [35]. The taxonomy for driving automation (see Figure 2.7) can be roughly understood as Level 0 without automation; Level 1 hands-on/shared control; Level 2 hands-off; Level 3 eyes

closed; Level 4 mind off, and Level 5 steering wheel optional. L1 performs either longitudinal or lateral control, unlike L2 which performs both at the same time. Examples include adaptive cruise control (ACC) and lane-keeping assist (LKA). L2 represents partial automation such as Nissan Pro Pilot Assist. Level L3 performs object and event detection and response (OEDR). At this level, the driver does not need to pay attention to the driving task. He automatically receives alerts when he needs to take back control of the vehicle. L4 and L5 are the higher levels where the vehicle can handle emergencies autonomously and the driver can concentrate on other tasks. The industry knows how to manage L1 and L2, so the engineering part is not so challenging for SOTIF. At the first level, accelerating and braking are easy and simple functions to understand what is their implication. SOTIF becomes a concern and a challenge for other levels of driving automation, because we ask how many scenarios are covered when the car is highly automated since testing real data may take too many miles, so we have to find measures to replace them.

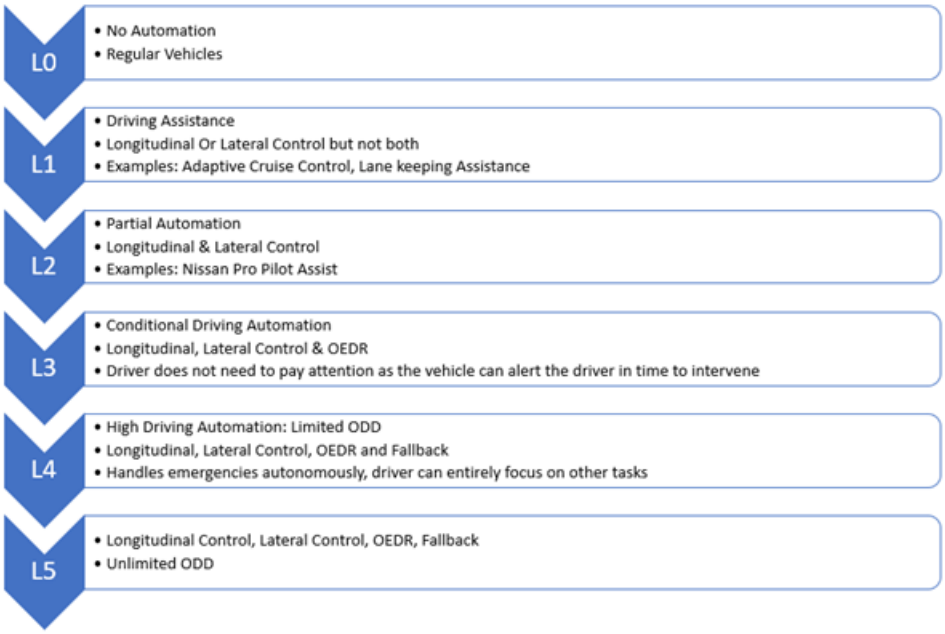

Figure 2.7: Taxonomy for driving automation.

Traditional FuSa always assumes that something goes wrong in the system and is not safe, and SOTIF thinking is earlier than FuSa thinking because it asks the question of whether a safe defined function for the client could fail and does not work as it is supposed to. With ADAS technologies, SOTIF comes to the rescue by giving engineers the ability to design systems that cannot harm or endanger people. As the automated levels arise, it is not so easy to ignore any implications a function might have and that's where the SOTIF idea comes in to

ask if that function as is defined is safe or not. Malfunctions become misuses and are not defined in an Excel spreadsheet or a table of cases to decide whether it is safe or not. With ADAS/AD functionalities, the list seems endless, taking the case of traffic variations or the example of emergency braking assistance. If we want to make it safe according to ISO 26262, implementing emergency braking must not harm anyone. It is supposed to brake before hitting an obstacle so that all participants such as bicycles, pedestrians, or cars are safe. We can now define the function and ask when to start braking and how strong. In some situations, this may not be appropriate, for example, if we are braking hard and someone is behind they could also hit us. Thus, the action defined for braking assistance which has as its main idea not to hit someone in front is perfectly correct, but it depends on the comfort we want to give to the driver. Another topic comes for false positive and false negative braking while assuming we have the perfect environment, we can handle these situations as well. It is important to equalize all functions by updating SOTIF to minimize the overall risk. If we find out that this minimization and the updated risk of the improved functions are not sufficient, then additional measures are necessary. SOTIF makes the system safe by asking the question is it safe in the first place and comes out with the result showing the risk and proposing the engineer take it or optimize it further.

2.4 Hazard And Risk Analysis (HARA) technique to evaluate safety

Safety is the absence of unreasonable risk of harm. By risk, it is the probability of having harm with a certain severity. The hazard is the potential source of unreasonable risk of harm, for example, if the system software has a bug that can potentially cause an accident, the software bug will be the hazard. Sources of hazards can be mechanical, such as an incorrect assembly of a brake system can cause a premature failure. They can be electrical, such as faulty internal wiring leading to a loss of indicator lighting. They can also be hardware (failure of computing hardware chips used for AD), software bugs, or even bad or noisy sensors or inaccurate perception. Hazards can also arise due to behavioral incorrect planning or decision making. It is also possible that the fallback fails by not providing enough warnings to the human driver to resume responsibility, or maybe the AD system gets hacked by some malicious entity.
Hazard And Risk Analysis (HARA) is the process commonly found in ISO 26262 based projects. It is an effective systems engineering analysis technique to ensure functional safety. When it comes to SOTIF, the process is quite similar to FuSa. Since SOTIF deals with function limitations, no malfunctions are considered in the scenarios. SOTIF also starts by analyzing the system regarding hazards, which is so much alike between these two standards. The first step is basically

the same thing. In SOTIF, HARA is applied by finding root causes and covers reasonably foreseeable misuses that are not covered by ISO 26262. The root cause is something SOTIF treats unlike ISO 26262. SOTIF considers the condition of the environment of the overall situation that triggers the system to do something inappropriate. We can take the example of an antenna failure, which is a malfunction treated by FuSa. The loss of communication is a case of misuse treated by SOTIF. For SOTIF, it is important to limit the list to just the failure scenario at the function level without defining the source of the failure. This is why SOTIF comes as a complementary standard for FuSa. The operational scenarios in which the functional faults are to be analyzed are defined. These scenarios shall be a combination of the below three factors:

- What is the vehicle doing (accelerating, decelerating, etc..)?
- Where is the scenario happening (type of road and weather conditions)?
- What is the environment of the AV (obstacles, construction zone, etc..)?

This activity captures the various scenarios and environmental conditions that might be hazardous for the AV so that the safety risks associated with each use case can be analyzed.

2.5 Formalization of safety requirements

Formalizing the requirements has the potential to dramatically reduce the probability of system failures during the development of automotive embedded systems [36]. We can have three levels of languages.

- *Natural language* that is a language anyone can use. They are easily understandable and they evolve naturally. Dealing with safety requirements, experts try to impose some order on them so they can define their way of ensuring safety. Many disadvantages arise from using natural languages. They are full of ambiguities, which people deal with by using contextual clues. In order to make up for ambiguities and reduce misunderstandings, natural languages employ lots of redundancy [37]. As a result, they are often verbose.

- *Formal language* is designed by engineers for specific applications. It is meant to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of the context. For example, the notation that safety experts use is a formal language that is particularly good at denoting requirements. Formal languages must have strict rules about syntax. They are less redundant and more concise. The main hurdle faced in using formal languages is the ability to learn how to use them. Users

will have to learn and follow a strict guide to understand fully the language's grammar rules. The rules being more formal than with natural language usage, it usually takes longer to read a program because the structure is as important as the content and must be interpreted in smaller pieces for understanding.

The software safety requirements sometimes are non-formalized and expressed in natural languages. At the software architectural design phase (seen in Figure 2.1), many use semi-formal modeling language (such as UML [38, 39]) to support the design and analysis of safety. However, at the software unit design phase, tools like Simulink are effectively used for prototyping, simulation, and code generation [40]. Semi-formal languages impose strict grammatical rules but do not always give a unique semantic interpretation. Formal languages require to have a strict semantic interpretation for each rule.
Four ASILs are defined in ISO 26262: ASIL A, ASIL B, ASIL C, and ASIL D, where ASIL A is the lowest safety integrity level and ASIL D the highest one [41]. ISO 26262 recommends formalization and semi-formalization techniques to determine higher ASILs for the hazards. Once the risk parameters are determined, ASILs will be assigned to the hazardous event from the standardized matrix provided in the ISO 26262 standard.

In this thesis, a new approach for the formalization of safety requirements is introduced, targeting the demands of safety standard ISO 26262. By following the proposed approach, we meet the obligations of ISO 26262 to write e.g. unambiguous, consistent, verifiable, and complete requirements. The formalization we follow was built in two steps, first by defining an operational interpretation by providing a unique transformation from the rules into C code. Second by providing a unique logical interpretation by transforming the rules into first-order predicates. The operational interpretation is used to derive monitors and study inconsistencies by running code on different scenarios. We use Xtext technology [2] to define the syntax of the language. By using the proposed approach, it becomes easier to produce proof by providing a formalization of each of the steps in the implementation of safety as required by ISO 26262. We accredit the goals in ASIL-D highly recommended by ISO 26262 to achieve formal safety requirements.

2.6 Model-Driven Engineering (MDE)

Model-Driven Engineering (MDE) is a software development methodology that focuses on creating and exploiting domain models, which are conceptual models of all the topics related to a specific problem [42]. Hence, it highlights and aims at abstract representations of the knowledge and activities that govern a

particular application domain, such as in this thesis, is the Safety domain. The MDE approach has many advantages such as:

- increasing the productivity by maximizing compatibility between systems,
- simplifying the process of design via models in the application domain, in our case Safety domain, and
- promoting communication between individuals and teams working on the system.
- promoting robustness to changes. When a change in a requirement occurs, parts that are not affected are reusable. However, an update is required for the changing parts, which requires tool support.

Actors as important as the Object Management Group (OMG) [43] and the Eclipse "Eco-system" propose a set of techniques and/or tools based on models for software development. Some of the better-known MDE initiatives is the Eclipse Modeling Framework (EMF) for programming and modeling tools which we use in the thesis [44]. This framework allows the creation of tools implementing the Model-Driven Architecture (MDA) standards of the OMG [45]; but, it is also possible to use it to implement other modeling-related tools. The idea of using MDE is to present levels of abstraction for developing systems. The main activity is to design models instead of directly developing code and following a more precise methodology. This helps define steps before attacking the development part, and the developers are alerted of what is done at each step and how by detailing the methodology. Capturing the requirements is an important thing to take into account while defining the abstraction. Use cases come to detail how to achieve the abstraction and describe the problem [46]. Feature oriented-diagrams also play a key role in that [47].

At each stage of an MDE process, quality management information could be integrated, such as verification, validation, and test case generation as promoted by the SW V-model as shown in Figure 2.1. Verification is to verify if a more concrete model does not break the specification promoted by its abstract specification model. A validation can allow system developers to instantiate prototypes from intermediate models to test their functionality before full system implementation. Automatic test case generation can produce result scenarios, thus making it possible to test its actual implementation, for example in [48].

An MDE process should thus define:

1. levels of abstraction, abstract syntax, and concept domains for each level;
2. the refinements and additional information in the lower level of abstraction that is represented in a concrete syntax;

3. how code is generated for the modeling language and how to deploy the code represented in semantics;

4. how can a model be verified and validated following the upper-level models.

Metamodeling technique [49] allows methodologists define precisely a class of models or domain concepts that define level of abstraction. It defines a modeling language by specifying its abstract syntax, eventually along with its semantics. Concrete syntaxes come along with this abstract syntax to give the user the capability to view and modify models using different modeling notations. *Model transformation* technique [50] allows methodologists to define relationships between models, and to add refinements on them. It gives the possibility to verify concrete models. *Code generators* technique is needed to map a model to some textual files, that constitute the operational semantics.

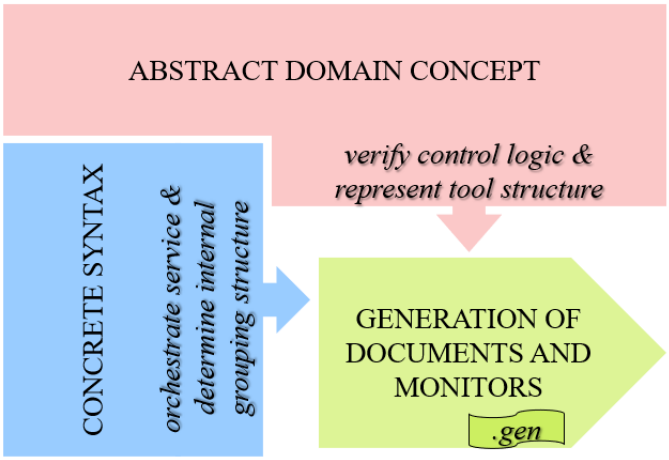

Figure 2.8: Language development structure defining abstract and concrete syntax with artefacts generation.

A significant issue is the current lack of a tool-independent solution and proper modeling notations to fully specify how to generate test cases from models, how to perform validation and deployment, and how to describe those models.

The standard MDE uses artefacts for defining a language's syntax (using *metamodels*) and its operational semantics (using *model transformation*) as seen in Figure 2.8. The language's syntax helps for formalizing the AVs' environment and requirements using Xtext technology [2]. The operational semantics are defined to generate monitors to verify the inconsistencies between the rules and check ambiguities in violations (using *code generators*). They are defined using Xtend framework [51].

All of these technologies used in the thesis to define the MDE process are grouped in the Gemoc initiative integrated with the Eclipse Gemoc Studio [52].

We detail these technologies in Chapter 3 Section 3.3.1. Examples of MDE processes and applications are provided in the following Chapters and appendix A.

2.7 Model-Driven Architecture (MDA)

Model-Driven Architecture (MDA) is a software design approach for the development of software systems. It is an OMG initiative that provides a set of guidelines for the structuring of specifications, which are expressed as models. MDA is a domain engineering that supports the MDE of software systems. It separates the specification of the operation of a system from the details given by defining a specification architecture. MDA is entirely implemented by means of models and model transformations.

To promote MDA, OMG including many US organizations joined to create the Unified Modeling Language (UML) standardization process. It was born from a unification concern regarding object-oriented modeling languages and was spread in many interesting languages (OMT, Booch, etc.). UML is a set of coherent modeling notations that can capture requirements using case diagrams and data structure using object diagrams with constraints. Many motivations to use UML are documentation, communication between actors of the project such as architects and developers, prototyping, abstraction, code generation, UML was first proposed as a universal language, and later new extensions were proposed to add notions to the UML. MDA community opted for a Domain-Specific Language (DSL) and OMG proposed new standards for modeling languages definition.

2.7.1 Metamodeling

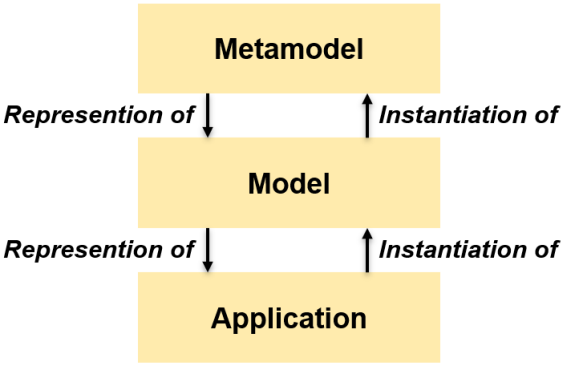

Figure 2.9: MDE process

As seen in Figure 2.9, a metamodel is a model about models. Many existing definitions may be found, but in the context of this thesis, we will consider metamodels as the specifications of the abstract syntax of a language. The abstract

consists of having a vocabulary and properties to each domain concept or object, and relations between them that constitute the taxonomy. A metamodeling language that has an abstract level has certainly a metamodel. In the context of this thesis, we use Eclipse Modeling Framework (EMF) [44] that permits us to describe the abstract syntax of modeling languages. EMF goes beyond storing models by applying code generation and generating storage artefacts for models with a specific API to make it possible to manipulate changes in the model. The language in which the metamodel is defined is called *Ecore*. *Ecore* is an essential part of EMF. Your models instantiate the metamodel, and your metamodel instantiates *Ecore*. The metamodel is the *Ecore* model of your language. In EMF a model is made up of instances of *EObjects* which are connected. An *EObject* is an instance of an *EClass*. A set of *EClasses* can be contained in a so called *EPackage*, which are both concepts of *Ecore*. *EClasses* can have *EAttributes* for their simple properties. These are shown inside the *EClasses* nodes. We can take the example in Figure 2.10 taken from [2] where we show *Model*, *Type*, *SimpleType*, *Entity*, and *Property* as *EClasses*.

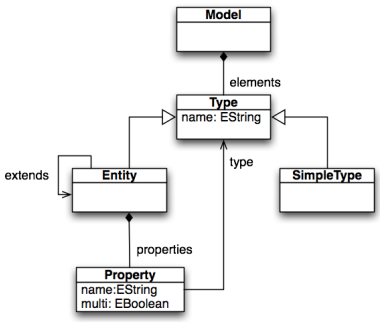

Figure 2.10: Metamodel in EMF taken from [2].

2.7.2 Model transformation

Model transformation is the second important technique of MDE. It makes it possible to build bridges between modeling languages for dedicated purposes. They can cover a full exploration and manipulation of models. To define the language and cover the concepts of models, we need to specify their concrete syntax. A concrete syntax can be textual to describe particular representations of models, graphical which uses graphical icons (e.g., tree-like) to show the elements of the model and relations among them or both of them. Sirius [53] and Xtext [2] can specify the graphical and textual concrete syntax respectively. Sirius could be used in the concept of this work. It is a visual language that describes a set of visual sentences given by a set of visual elements. This improves readability and usability. In this thesis, we use Xtext for the concrete definition of the models. It

is possible to have several different concrete syntaxes for the same abstract syntax [54]. Moreover, it is possible to apply the same concrete syntax for different abstract syntaxes since the concrete and abstract syntaxes are separated. Xtext relies heavily on EMF internally and uses EMF models as the in-memory representation of any parsed text files. It acts as an interface between the instances of the concepts and the domain user is supposed to produce them.

2.7.3 Code generators

To define a language we need to specify semantics that results in artefacts. An example of such an artefact is program code, which can be compiled into an executable file. To do so, there exist several code generators from models. Xtend technology helps generating artefacts in form of a monitor and verification engine, and even documents using generators. Xtend is a statically-typed programming language that translates to comprehensible Java source code. The benefits of code generation are the following :

- **productivity:** with code or text generation, we only need to write the generator once. It saves time as it can be reused as many times as we need.
- **simplification:** the source of truth is the abstract description in the metamodel and not the code. Instead of analyzing and comparing the whole generated code, the engineer can replace the developer by only accessing the model.
- **portability:** targeting different languages or frameworks is not a burden anymore. Changing the generator is a solution; for instance, with a parser generator, C#, Java, and C++ parsers are generated with small modifications. The same abstract syntax can be used to generate different types of artefacts.
- **consistency:** We systematically apply generation rules so consistently across a model. The generated code or text follows the user description and applies the same methods defined, such as creating domain concepts and defining naming rule matches. The quality is consistent throughout the whole model as it is done systematically. With written code, the industry distributes the tasks to different engineers and developers. With generated code or text, they are replaced with one engineer who defines the rules and the related transformations.

2.8 Domain-Specific Modeling Language (DSML)

A Domain-Specific Modeling Language (DSML) is a language that supports solutions to crucial problems in a given dedicated domain using models. As prescribed

by the DSL community, MDE promotes the usage of various "small" dedicated languages. In this thesis, as we are dealing with the Safety domain, using DSML intends to raise the level of abstraction closer to safety users' domain understanding. Unlike a General Purpose Language (GPL), such as Java or C meant to apply to multiple domains, a Domain-Specific Language (DSL) offers the end-user the ability to express their needs in terms of the problem domain rather than of computational solution [55]. DSL provide a notation tailored towards an application domain as they are based on models of relevant concepts and features of the domain [14]. They give an expressive power to model the requirements more easily and simplify the development of applications in specialized domains.

DSL offers important advantages over a GPL [56]:

- Abstractions: providing abstraction level to represent domain concepts from the application domain;
- Concrete syntax: providing natural notation for a given domain and avoiding the syntactic clutter;
- Error checking: enabling building static analyzers that can find and report more errors in a language familiar to the domain expert;
- Optimization: creating opportunities for generating optimized code based on domain-specific knowledge;
- Tool support: creating opportunities to improve any tooling aspect of a development environment, including editors, debuggers, etc.; domain-specific knowledge can be used to provide the smarter tool support to developers.

The use of MDD techniques and tools is considered a viable approach to address accidental complexity [57]. MDD transforms the explicit models into other lower-level models considered as development artefacts using model transformation. This transformation approach deals with the complexity of large-scale problems enabling prototyping, simulation, and validation and verification techniques [16].

A DSML provides interfaces for activities such as model execution. It also provides a structural way to define a concrete and abstract DSL through the use of a meta-modeling environment. It facilitates rapid development but is at the same time somewhat restrictive. It produces similar results to the IDE design style, although it is significantly more sophisticated. Different tools and platforms are now being defined to support DSL implementation and processing, such as, Microsoft DSL Tools [58], Generic Modeling Environment (GME) [59], and Epsilon [60] based on GMF/EMF [61, 44].

2.8.1 DSL stakeholders

Many DSL stakeholders exist in the application of the SW language.
The *Language Engineer* manages the implementation and design of a functional SW language. They are involved in the language specification, implementation, and evaluation, as well as in providing templates and scripts [62].
The *DSL User* or *Domain User* is a person who uses the DSL to create applications [62].
A *Domain Expert* is the person involved in the language development process. He is an expert in systems software development. He is a person with special knowledge or skills in a particular area of endeavor.

DSLs are usually built by *Language Engineers* in cooperation with *Domain Experts* [57]. *Domain Users* will use the DSL and are considered the target audience for it. Although *Domain Users* know the domain, they are not necessarily as experienced as *Domain Experts*. They may also not have the experience of *Language Engineers* in working with languages. Thus, it may turn out that the language is valid by construction for *Domain Experts* and *Language Engineers*, but not necessarily for other *Domain Users*. Neglecting domain users in the development process can lead to a DSL that they are not able to work with.

2.8.2 DSL life-cycle

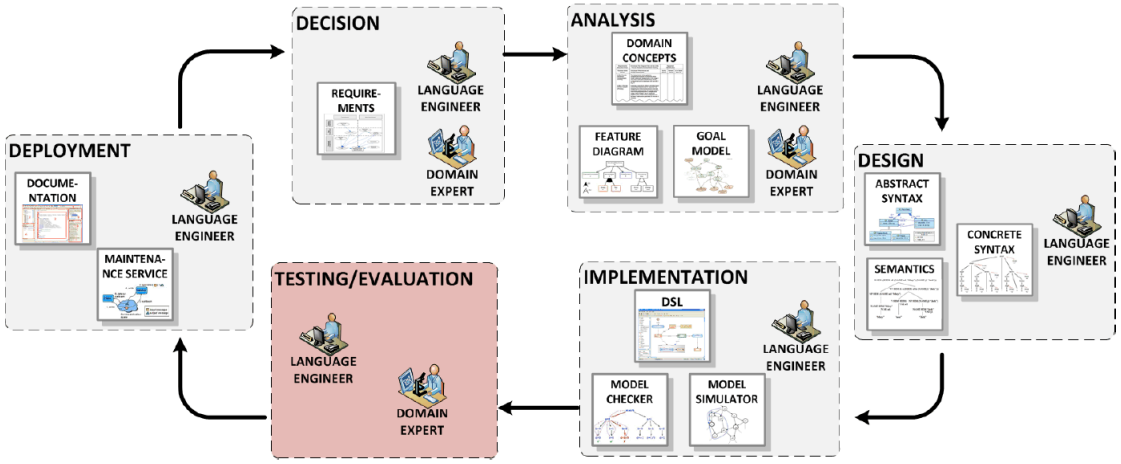

Figure 2.11: DSL life-cycle taken from [3]

For the tool development in this thesis, we follow a DSL life-cycle detailed by Völter [57], Mernik [14], and Visser [63]. The following list constitutes the DSL or DSML development phases presented in Figure 2.11 taken from [3]:

- The **Decision** phase is the "when" part of DSL development, while the remaining phases are the "how" part. Its purpose is to identify the need

for a domain DSL and its validity, which includes justifying that the effort of investing in its creation can be compensated. To make the decision, domain experts and Language engineers should discuss domain requirements following DSL development patterns.

- The objective of the **Analysis** is to define the domain model with DSL support taking into account the terms and expressions related to the dedicated domain that will be explored. During this phase, domain experts help language engineers define the domain concept description and the feature models, and the functional and technical requirements. The analysis phase mainly produces a domain model which is the metamodel, representing the common and variable properties of the system in the domain [56]. The domain model will be used to facilitate the creation of configurable architectures and components.

- The **Design** phase is a language engineer task in which he formalizes his abstract syntax and defines representations and rules composition. The engineers also specify the semantics of these rules. Traditionally, all car software was written manually based on detailed low-level software requirements using natural languages. Today, the trend is to replace programming with higher-level domain-specific languages. The principle is to model the necessary functional behavior by simply adding library blocks and linking them. The specification model obtained is then simulated by the designer to check whether its behavior conforms to the specification. Once the specification model has been validated, there are two ways to integrate it into the car component. The oldest way is to use the model as a software specification and manually write the code corresponding to the model. This method can be error-prone and time-consuming. This is why the preferred method today is to transform the specification model into a design model that can be used to automatically generate the target code. The formal nature of DSL is previously described in Section 2.5.

- The **Implementation** phase integrated the DSL artefacts with the infrastructure and all the necessary implementation to the platform transformations. A DSL can be implemented by different approaches, for example, model checkers and simulators in order to help the interpreter or the modeler validate the specified model and requirements.

- The **Deployment** phase is the last phase where domain experts deliver documentation to validate and understand the software.

- The **Evaluation** that should include the verification (testing if the right functionality is provided by the DSL) as well as its validation (testing if this DSL is right for its users) presented in [3].

2.9 Overview of existing safety solutions

Many existing solutions above have come up with new approaches that can be validated offline without requiring the use of large test traces extracted from hours of observations. One of these approaches is called Responsibility Sensitive Safety (RSS [12]) and has largely inspired our proposal. S. Shalev-Shwartz, S. Shammah, and A. Shashua have created a white-box approach for the interpretation of safety assurance requirements. They have developed redundant sensing systems in a complex environment. Besides the fact that a software change will require new data collection, they lack a fundamental property: interpretability. In case of an accident, we need to know why it has happened and who is responsible, and possibly to prove that the autonomous driving systems took all the precautions to prevent it. This model does not aim to guarantee that a vehicle will not be involved in an accident, but rather to make explicit the assumptions and driving policies under which it can guarantee the safety of the vehicle. This model is also parametric in the sense that it needs some values that, as the authors suggest, should be discussed between regulatory bodies and vehicle manufacturers. However, this approach does not identify precisely the events involved and does not include environmental conditions such as weather.

Sensor uncertainty dilemma has been handled using probabilistic models with formal specifications [64]. A rule-based strategy was also designed to evaluate sensors' dependability [65]. Yet, the main problem of autonomous driving systems perception is still not explored. In this thesis, we give the user the capability to add parameters and specifications relying on the sensors and the environment, to examine ambiguities.

An approach is adopted in [66] where they address the potential conflicts between safety and security properties. They considered safety and security properties during the design phases of a platooning autonomous system: from textual specification, modeling, 3D simulation, embedded software to early prototyping. This is interesting to combine the proposed approach with their framework to automatically generate the two domains and evaluate them.

Many of the existing solutions use MBSE approaches, but most of the studies are done to create and generate new scenarios or review ones designed by other experts [67]. They ease the scenario creation process but do not formal safety rules descriptions. Furthermore, a SceML study was carried out [67] 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.

Another solution is a modeling and simulation environment called STIMULUS developed by Argosim that has been acquired by Dassault Systèmes [68]. This In-

dependent Software Publisher (ISV) develops and tests requirements in real-time and helps system integrators and their suppliers in critical areas such as transport, nuclear, and medical systems. STIMULUS also highlights inconsistencies and ambiguities in requirements by validating them from the specification phase. It also allows for the fast detection of incomplete or underspecified requirements. Developers used MS Word (informal), SysML, and Papyrus. This solution is proprietary and license-based. Unfortunately, what the autonomous driving world needs is a standard accessible to all, open-source, with a formal semantic. Similar solutions to this exist, however, some of them do not offer a language subset that can be used for scenario description. We can mention Measurable Scenario Description Language (M-SDL) created by Foretellix [69]. Despite being released openly (under Apache License, Version 2.0), the modeling and simulation tools are proprietary solutions. In order to achieve standardization in the industry, we need a whole ecosystem that is open-source.

Moreover, the language has some major downsides. For example, in M-SDL there is no way to specify important details such as the key characteristics of a sensor (sample rate, accuracy). This is important to understand the assumptions made about the environment and how precise and accurate the decisions can be. However, it is good to praise the good advantages of M-SDL. Its toolkit is capable of generating combinations of scenario variants along with monitors to check and track scenario coverage. In other words, with the simple description of one behavioral scenario, the specification may represent hundreds of thousands of scenario variations, each with slightly different conditions. Each of these scenarios will be tested including edge cases. The tools also allow us to monitor and measure the coverage of the autonomous functionality critical to proving autonomous vehicle (AVs) safety.

In [70], the authors presented an Event-B model [71] for the exterior lighting system case study. Their model takes into account all the requirements and has been verified by proving a large number of properties. It is interesting to look in detail at the requirements and try to adapt with the AV domain. They prove that formalization leads to identify several small ambiguities in their requirements, and is an effective technique to discover defects early in the software development process. The Event-B model [72, 71] presents a formal method of verification of transport systems is adopted, famous for its use by the RATP [73] on the verification of trains. It would be interesting to combine these formal methods to ensure that trains run safely within a given network in the use of AVs. In [74], Finkelstein introduced the notion of inconsistency management and discussed the use of first-order predicate logic as a formalism to introduce logical contradictions. In [75], the authors use a propositional logic approach, similar to [74]. Both studies present limitation of insufficient expressiveness and complexity associated with expressing and translating software models in a logical formalism [76]. In [77], A. Kreutzmann, D. Wolter, F. Dylla, and J. H. Lee have developed a software tool based on the concept of proof-carrying code. It enables software developers'

verification for a formalization of collision regulations. This was done based on Spatio-temporal logic that led to capturing complex navigation concepts with software verification.

An MBSE approach is proposed by [78] to address safety and security. The model is based on three viewpoints that enable the expressive language to assess the properties of the system architecture. Those standards cover additional safety perspectives but not for fully AVs since their environment is more complex and complicated.

2.10 Conclusion

The need for automated vehicles (AVs) necessitates the establishment of a safety assessment methodology for road approval authorities. Therefore, we have presented an overview of safety approaches and detailed safety international standards, FuSa and SOTIF, to evaluate safety. SOTIF can be considered as a complementary standard for FuSa taking into account the misuse of environmental conditions. We detailed the HARA, a common technique used in both standards, that captures various scenarios that could be dangerous for AV so that the risks can be analyzed. We described the formalization procedure recommended by the ISO26262 standard to define rules and detect inconsistencies. SOTIF adopts a scenario-based assessment, using real-world data or generated to use a scenario database with scenarios that an AV might encounter when operating in real life. We mention the MDE methodology that focuses on the creation and exploitation of domain models. A level of abstraction and refinements are defined using a concrete syntax. A code generator is helpful to specify the semantics that generates artefacts represented in monitors, documents, and verification engines. The DSL life-cycle deals with the safety domain by enabling Test-Driven Development for safety and focusing on detecting rules inconsistencies and ambiguities in scenarios. It has significant advantages over GPL, such as optimization by creating generated code based on safety knowledge and supporting that knowledge by creating a gateway to the development domain.
The proposed methodology is based on what is previously mentioned to evaluate the safety, focusing more on SOTIF standard. It enables the user to formally specify requirements to check safety by generating a monitor, a verification engine, and a document. This formalization process helps produce proof as required by ISO 26262 and accredits the goals in ASIL-D highly recommended by ISO 26262. We mentioned what are the existing safety solutions and how our proposed framework can be an extension of these works. The tool created is an iterative approach that consists of improving the requirements after evaluating the safety of the generated code.

Chapter 3

Proposal for an Extensible Platform for Safety Analysis of Autonomous Vehicles (EPSAAV)

3.1 Introduction

In this thesis, we want to take into consideration the aspects previously identified in Chapter 2. MDE [42] promotes the use of models in the software life cycle. The models consist in defining a level of abstraction which helps the developer to concentrate on the model which constitutes the core business, and gives him a better understanding of complex systems. Another activity is concretizing the model by improving it to integrate details of the final system. MDE has many advantages like providing means to automate the concrete mechanism, and formalizing methods using formal specifications. Another advantage is involving automatic code generation replacing writing code manually. We rely on DSML

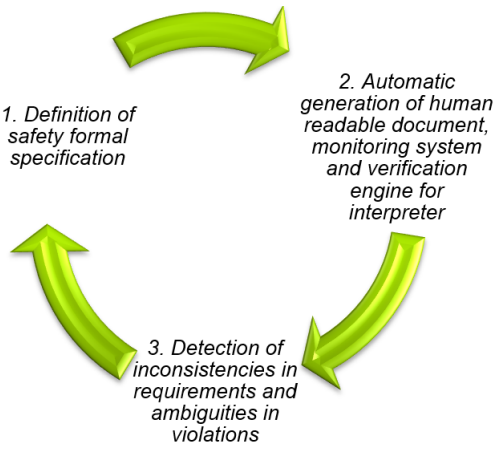

Figure 3.1: Proposition cycle.

with the principles of MDE. We present a novel model related to the safety of AVs. A simulation platform is designed to analyze the environment and the trajectory of AVs within a given ODD.
Figure 3.1 shows the three sequential phases that constitute the proposition cycle of the approach.

1. This platform depends on model-based systems and includes the AV's environment, safety rules and their priorities, and execution scenarios. This is considered the first phase of the cycle in Figure 3.1. The safety expert needs to define all the environments and the conditions in libraries. He also needs to describe the requirements in a rule-based planner. Building libraries help to reuse the same constructs later on.

2. Once all the libraries are ready and instantiated using the formalization tool created, our platform helps produce a human-readable document, a monitor, and a verification engine. They are generated and constitute the monitoring system.

3. Using the generated systems, the monitor is fed to a simulator to visualize breaches and detect ambiguities from the formal specification, and the verification engine is analyzed by a SAT solver to detect inconsistencies within the defined rules that seemed valid when expressed in a natural language but have no actual logical interpretation.

Therefore, this platform helps to reevaluate the existing rules in two ways: either by reconsidering rule priorities or by proposing new rules to be integrated into the existing safety model. The validation and verification of the generated rules follow a process based on the formal rules applied on real or simulated scenarios. The developed platform helps the user define his formal rules. Thanks to this approach, we also enable the detection of ambiguities and inconsistencies in real or simulated scenarios.
It is important to differentiate the user process from the language development

Figure 3.2: Two-way views of the approach.

part. We present two perspectives of our approach in Figure 3.3.

First, we introduce general steps for user perspective in Section 3.2. We detail how the user can use our approach and what he needs to do to perform a safety assessment. Second, from a language engineer perspective detailed in Section 3.3, we describe the necessary steps to implement the DSML. We detail our metamodel and concrete syntax to generate monitors.

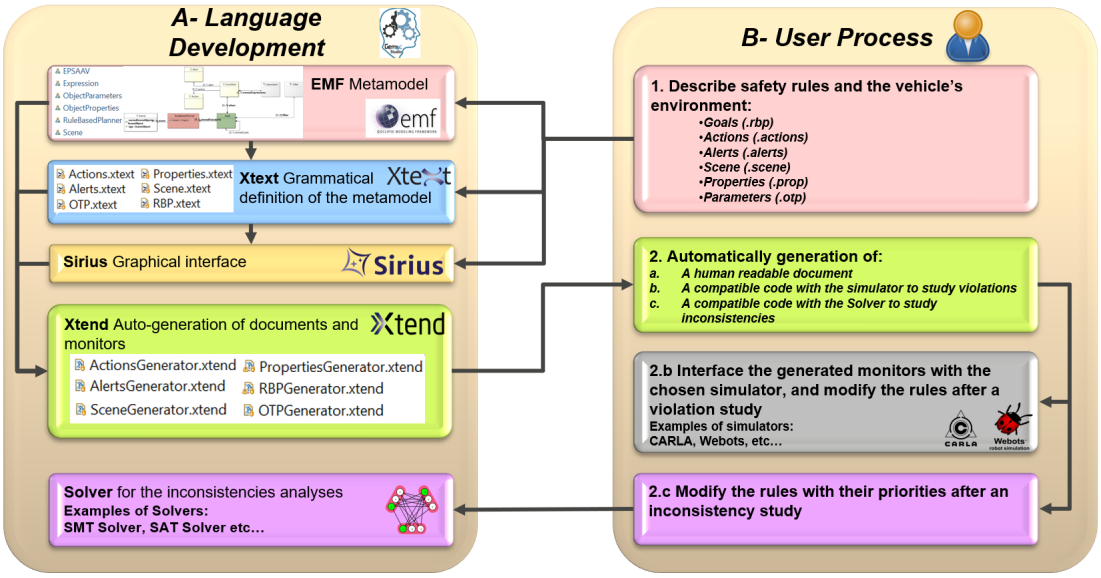

Figure 3.3: Two-way views of the approach.

3.2 User process

This Section focuses more on how the user can fill the files in the textual representation and create the model from EPSAAV.
Figure 3.3 presents the user process perspective. EPSAAV tool contains an **EPSAAV specifier** to illustrate and model all domain concepts in a formal grammar and help the safety experts formalize his requirements, and an **EPSAAV generator** that is responsible to generate the systems needed as seen in Figure 3.4. It represents a schematic diagram explanation for the steps that the user must follow to validate the application and make the violation and inconsistency studies possible:

1. First, he needs to describe the AV environment and rules by creating specific files in the application of the workspace. The six files constituting the formalized requirements and libraries have specific extensions defined in Section 3.3.3. One file consists of describing formal rules presented in Section 3.3.3.1, the one describing the scene is discussed in Section 3.3.3.2, two files are for parameters as seen in Section 3.3.3.3 and the properties shown in Section 3.3.3.4. Two files are for alerts and actions libraries addressed in Section 3.3.3.5. The engineer only has to follow the format proposed from the **EPSAAV specifier** that eases the structure and formalizes the input data.

2. The **EPSAAV generator** is used to produce resulting artefacts:

   (a) The generated textual document is meant to replace paper documents

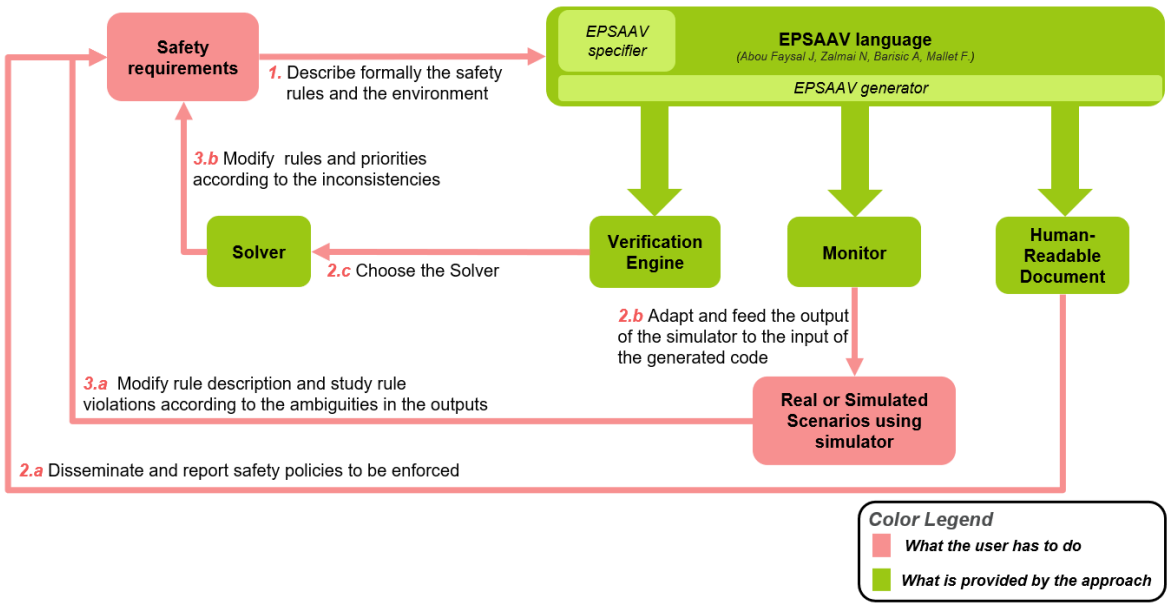

Figure 3.4: Schematic diagram explanation for the user process.

used in the traditional flow. Being able to generate it fully shows that our language has the right expressiveness. It contains the description of the requirements and serves as a reference for all safety engineers to report and disseminate safety policies to be enforced in Renault.

(b) The C-language monitors are generated and should be interfaced with the chosen simulator which applies real or simulated scenarios. Whenever the user defines generator-specific inputs, the EPSAAV language produces code faster than if it had been written manually. Those monitors give an operational semantics to the requirements presented in the textual document. Among the relevant simulators, we can cite public ones such as Apollo provided by Baidu [79], CARLA [80], Five AI [81], and Webots for automotives [82]. CARLA and Webots are very promising and are largely used by the automotive community. In our case study, we use *FusionRunner* debugger as a resimulation tool that debugs real and simulated scenarios since it is already integrated within Renault process. This part is more detailed in Chapter 4.

(c) Another generated code is useful to test inconsistencies in the rules. It is a verification tool that uses formal semantics defined by the safety expert to detect inconsistencies in the requirements. We have included a solver in the language so that the user does not have to interface it with the code. Among the relevant solvers, we considered SMT [83] and SAT [84] solvers as they are generic to compute satisfiability. In our use case, we deploy a SAT solver as the rules mainly consists of

Boolean invariants. This part is more detailed in Chapter 5.

3. The automatic generation of artefacts allows safety to be assessed and rules to be modified based on outputs. It is an iterative approach that helps safety experts proceed to a Safety Analysis of Violations and Inconsistencies (SAVI) which we define.

   (a) The generated monitor, that will be interfaced to a simulator, allows our user visualize all violations and study ambiguities. Accordingly more rules can be added or they can be modified to remove detected ambiguities.

   (b) The generated verification engine detects inconsistencies among the rules according to the formal semantics. It is used as a support to make sure that the safety rules have an actual (unique) logical interpretation. When there is no formal support, it is easy to write rules that seem correct but have no valid interpretation or have only one trivial interpretation when the system is safe if nothing useful happens.

3.3 EPSAAV language development

The objective of this Section is to describe the EPSAAV tool using DSML to support formal rules definition and artefacts generation. Figure 3.5 gives a first

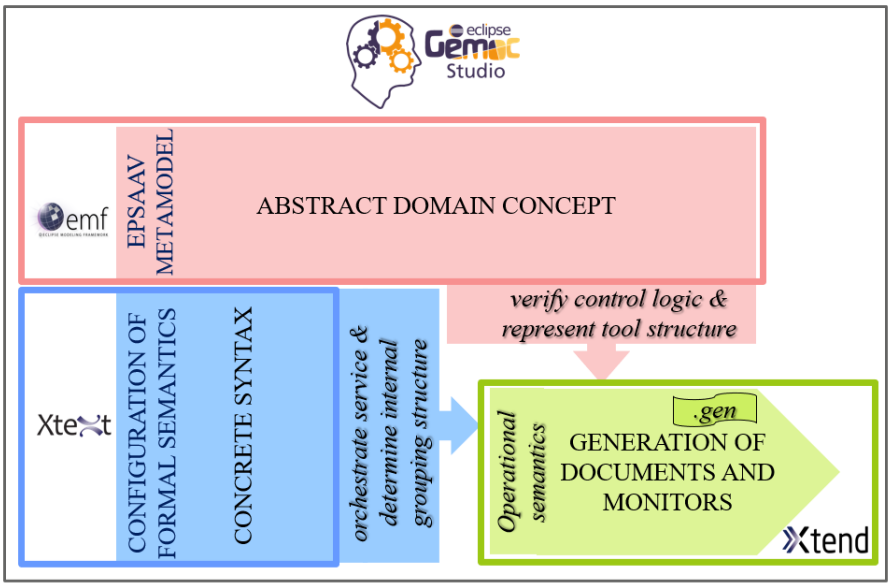

Figure 3.5: Overview of the used technologies grouped in Gemoc Framework.

representation of the overall language development process and the technologies used. It is the higher level description of Figure 3.3.

This view focuses on the model handled and its evolution during the approach. The EPSAAV metamodel proposed is defined by its abstract syntax, its grammar using concrete syntax, and semantics applying generation of documents and monitors in the language development.

Abstract Domain Concept represents the role of the actors in the EPSAAV developed. It verifies the control logic of the metamodel and represents the structure of the source code for the compiler to use. Concrete Syntax and the general sequencing of activities for configuration determine which text strings are accepted, and provide the internal grouping structure of the language. It indicates how the text is supposed to be grouped. EPSAAV and the configuration parts are considered as inputs for the main generation process. We can then ensure, assess, and validate the files' generation. Once the documents and monitors are ready, we proceed to a SAVI. This approach is not only limited to the Autonomous Driving (AD) function and the safety field. It is also intended to be integrated into the new set of methods, tools, and processes being deployed within Renault Group.

The implementation of the EPSAAV language is detailed in Section 3.3.2. To ensure the acceptability of this language, we need to make sure it has the right expressiveness and build support for analysis. We can manipulate this expressiveness at the concrete level described in Section 3.3.3. An automatic generation of a monitoring system (monitor and verification engine) assists the interpreter by triggering alarms and behaviors. The generation is depicted in Section 3.3.5. The monitoring system enables the user to detect inconsistencies thanks to the solver, and encounters ambiguities in violations of rules thanks to the simulator. We detail the SAVI process in the use case described in Chapter 6. After performing a SAVI, the user can modify the rules, which we call also goals, and repeat the same cycle to ensure the correctness of the autonomous behavior. The users activities of how they can use the platform are later detailed in Chapter 6.

3.3.1 Technologies used for the platform specification

Limited expressiveness makes it harder to express wrong things and facilitates verification. High expressiveness usually leads to undecidability issues. DSML provides solutions to describe rules using vocabulary close to the problem domain, while the actual code is (partially) generated [85]. Our framework uses Gemoc open-source tool based on Eclipse [86] to combine several heterogeneous technologies. The modeling environment focuses on design and validation problems in complex systems. One of them is enabling the evolution or creation of languages and models. It also integrates heterogeneous parts for different applications that work together to deliver a global service. Specification and simulation techniques aim to model and validate system design and architecture. They are combined with formal verification tools to describe and simulate what a system

should do. The use case in Chapter 6 is intended to illustrate this. The developed framework covers all aspects of a DSML, from abstract and concrete syntax to semantic operations, as shown in Figure 3.6.

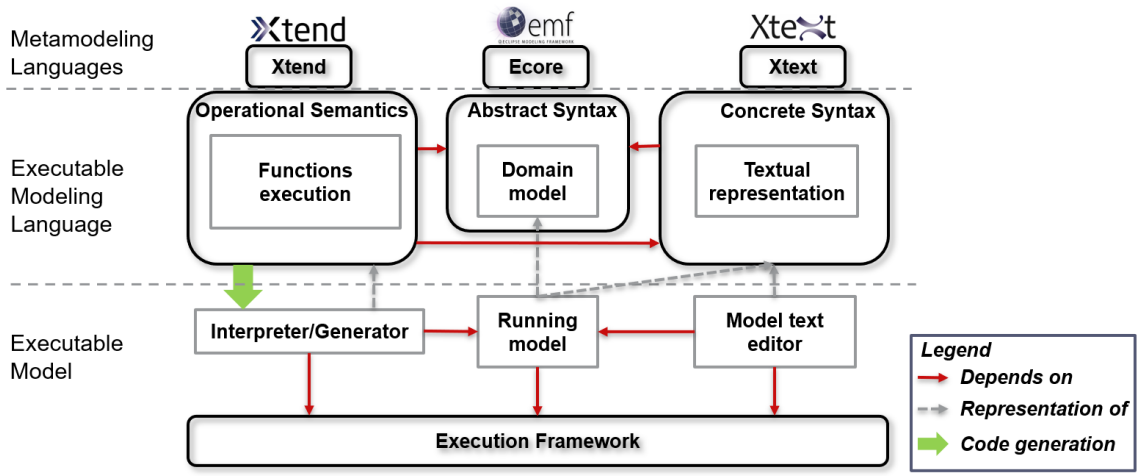

Figure 3.6: Gemoc Execution Framework.

We have started with the definition of the abstract syntax and the metamodel. It is based on Eclipse Modeling Framework (EMF) [87], which supports Ecore metamodel implementation. EMF consists of a graphical description of the metamodel. It is a framework and code generation facility that defines the model and generates implementation objects. It unifies the three important technologies: Java, XML, and UML. Its model is the common high-level representation that glues them all together. The EMF metamodel describes the objects and the relationships of the environment. Once we have the final libraries generated from EMF, this generative approach allows us to generate various concrete syntaxes by using Xtext artefacts [2, 88] as seen in Figure 3.6. It is interesting to note that the Gemoc framework generates an IDE with syntax checking using Xtext technology. Once the concrete syntax was processed, we need the operational semantics to assign behavior to each of the declarations in our DSL. To do this, we use Xtend [51], a programming language adapted to implement the execution semantics of Ecore metamodels. We are generating two types of artefacts: one for the integration with the AV's environment system (code for monitor generation, and code for verification engine), and the other one for the description of this environment and specifications (human-readable document).

The Gemoc framework is open source and integrates with other Eclipse-based tools. It features easy code generation and adapts when settings are changed so that it works properly as development continues. In other words, the language allows us to express the ODD in a common, non-ambiguous language in which we can express the scenarios that need to be handled safely by a vehicle to achieve "certification" as discussed by [89].

We use these technologies for the proposed language to shorten the development cycles and have an effective method of reasoning on these rules. The interesting thing about this approach is that we can adopt new requirements that we meet or change the existing ones by extending the language without changing the base. It is also good to note that the generated monitors reduce time costs for engineer development. Furthermore, safety experts can benefit from the tool that can reduce error implementation so the safety engineers could spend more time on rule specification. We describe in the following Sections Figure 3.7 for the abstract syntax (EMF), concrete syntax (Xtext), and operational semantics (Xtend).

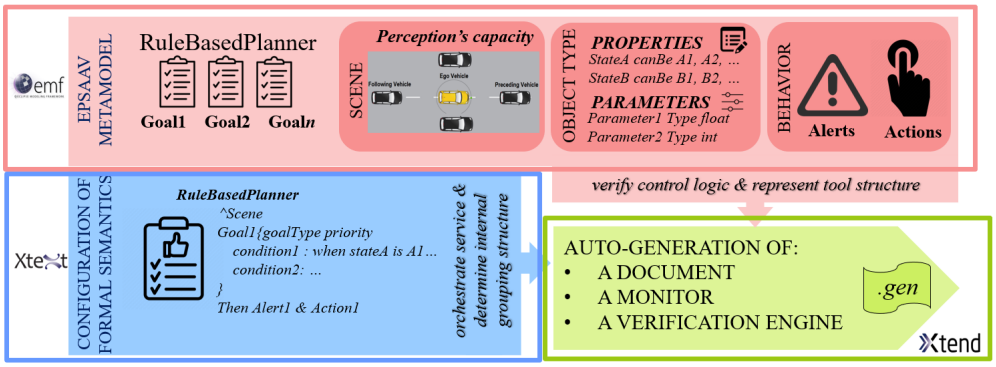

Figure 3.7: Defining EPSAAV language using Gemoc framework.

3.3.2 Abstract domain concept

Abstraction is a simplified selection of technical tasks that intend to reduce complexity by removing information that is not relevant or necessary. It allows the engineers to concentrate on core functionalities and to discard details by deferring secondary concerns. Models are examples of supporting abstraction in software engineering by hiding implementation details to the client software system and offering instead interfaces. Engineers can easily manipulate, communicate, and reason on models expressed in a given language as mentioned in Section 2.7.1.

Figure 3.7 gives a detailed representation of the three concepts or levels proposed. The first level represents the role of each actor in relation with EPSAAV. The metamodel consists of domain concepts to assess safety considering AV's environment. The first one in Figure 3.7 is the *RuleBasedPlanner*. It groups all the requirements defined by the safety expert. The *SCENE* domain concept consists of defining the perception capacity of the AV, such as it can perceive a Following Vehicle (FV) or a Preceding Vehicle (PV), or even both. The *OBJECTTYPE* contains all the *PROPERTIES* and *PARAMETERS* defined in libraries. Requirements depend on these libraries to evaluate safety. Finally, the *BEHAVIOR* raises alerts and actions when a rule violation occurs. We model these four main

Sections (*RuleBasedPlanner, Scene, ObjectType, and Behaviors*) in EPSAAV language at the abstract level. We use EMF presented in Section 3.3.1 to describe the metamodel.

The metamodel in Figure 3.8 shows the higher level of the domain concepts and their relationships and what we need to describe safety requirements.

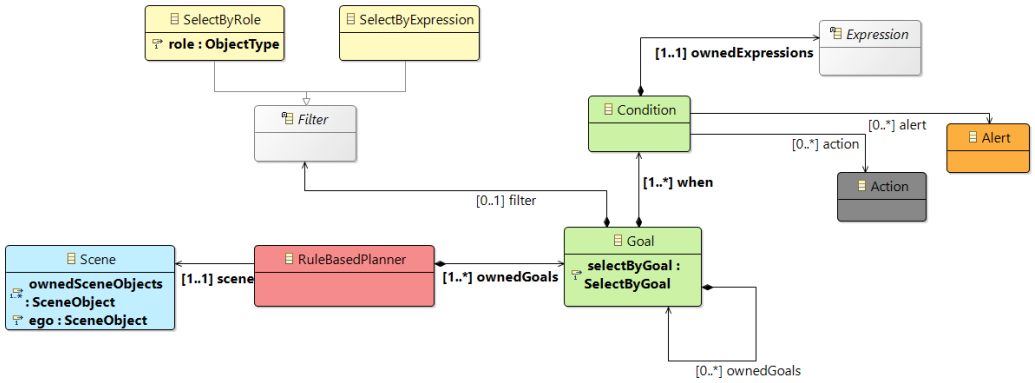

Figure 3.8: Abstract description of EPSAAV metamodel using EMF technology.

- *RuleBasedPlanner* (RBP) is the domain concept containing all the specifications or rules to evaluate safety. It is the central object of our metamodel. It is responsible for formalizing and specifying rules. It refers to a described *Scene*. It is composed of one *Goal* at least. By *Goal*, we mean the rule specified by the user.

- *Scene* domain concept is the perception capacity that describes the agents seen in the scenarios. The *Scene* is described in detail in Section 3.3.2.1. Each agent is the *SceneObject* containing parameter and property libraries described in Section 3.3.2.2.

- *Goal* domain concept is the safety rule or specification that is applied on a specific object type (*ObjectType*). The *ObjectType* can be either the autonomous vehicle (ego car), the preceding vehicle (PV), the following vehicle (FV), the pedestrian, etc. We created the *Filter* domain concept to be able to apply the *Goal* on a specific role object (*SelectByRole*). By role, we refer to an *ObjectType*. We also give the ability to filter by expression (*SelectByExpression*), which means to apply rules in certain conditions, for example in a traffic jam, or on highways, etc. It takes the same form as the *Expression* domain concept described later. Each *Goal* can contain multiple goals. Goals can be executed in parallel or sequential. The *SelectByGoal* is responsible for assigning the execution type to each rule, which means that

we need to know if a certain defined rule is sequentially executed with the others (it has a priority), or it is parallel. We detail *SelectByGoal* domain concept in Section 3.3.2.3. A *Goal* is composed of at least one *Condition*.

- *Condition* domain concept follows the form: **When** *Expression* **is True, then execute** *Alert* **and** *Action*. The *Expression* follows the logic expressiveness described further in Section 3.3.2.3. The *Alert* and *Action* represent the behaviors and are detailed in Section 3.3.2.4.

We will address in these Sections their detailed levels. We also note that for each new representation model of the domain concepts we present in the following figures, we try to hide the unnecessary domain concepts to be more precise and less complex. This is why the associations are shown as attributes with an association logo.

3.3.2.1 Scene and ObjectType domain concepts

Each driving task requires object and Event Detection and Response (OEDR). OEDR helps to identify objects around the ego car, that is the AV, detect events that occur nearby, and then react to them. There are three crucial parts of perception: (1) Static Objects (e.g.: road structure, traffic lights, and signs), (2) Dynamic Objects (e.g.: vehicles and pedestrians), (3) Ego module, or the AV module, that corresponds to internal parameters and properties of the AV, such as the speed, the acceleration, the safety distance with the seen objects, the Time To Collision (TTC), etc. For this reason, we created a scene specification, which is very important to describe the capacity of perceived objects around the AV. The *Scene* specifies the roles of these objects by attributing a name to each one, and their types as seen in Figure 3.9. It forces the system to have one *ego* car as a *SceneObject* that is referred to a predefined *ego ObjectType*. Each *ObjectType* is added to the *ObjectTypeLibrary* that has a version for update matters. Every *ObjectType* refers to a *PropertyTypeLibrary* that has a *version*, and is composed of a *ParameterTypeLibrary* (detailed in Section 3.3.2.2).

3.3.2.2 ParameterTypeLibrary and PropertyTypeLibrary domain concepts

In Figure 3.10, *ParameterTypeLibrary* is a library containing all necessary parameters of a given *ParameterType*. Each parameter can have a type (*PrimitiveParameterType*) such as *float*, *integer*, *boolean*, or *string*. These parameters are important to assess safety.

In Figure 3.11, *PropertyTypeLibrary* is a library containing all necessary properties of type *ListType*. Each *ListType* can have states (*StringValue*) that can be equal (=), less than (<), or greater than (>) a value, or it can be an interval of

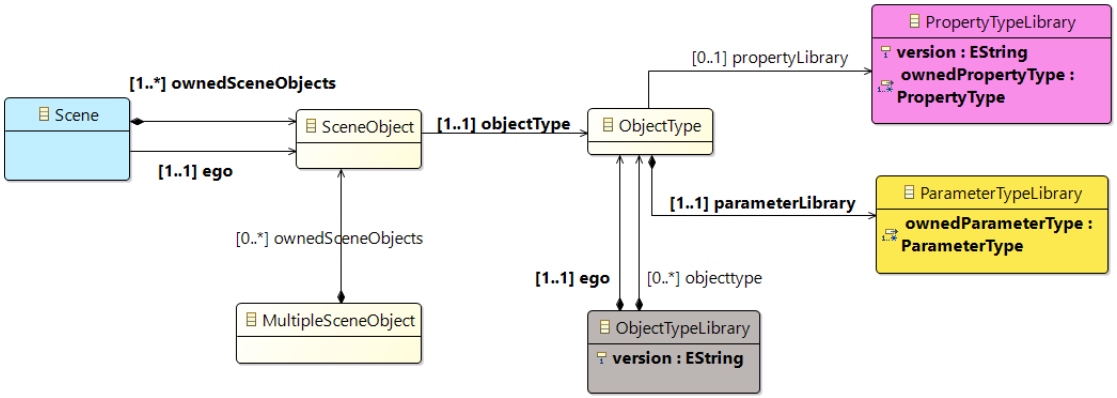

Figure 3.9: *Scene* metamodel composed of *SceneObject* referring to *ObjectType* containing a *ParameterTypeLibrary* and assigned to a *PropertyTypeLibrary*.

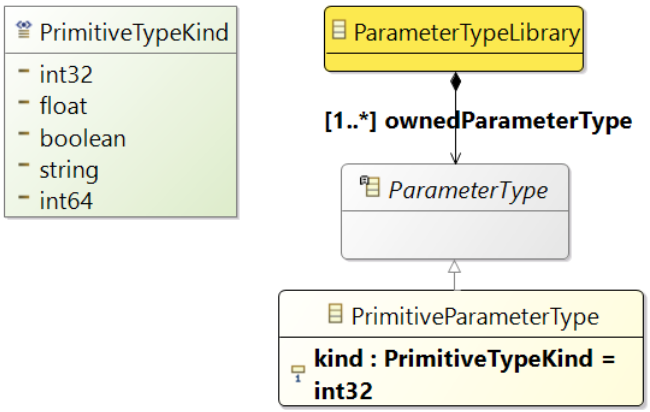

Figure 3.10: *ParameterTypeLibrary* containing parameters with types.

two values. This is why we gave the *StringValue* an *operator*, *value* and *value1* in case it is an interval, and a *unit* for the values.

For instance, we need the TTC value for the Preceding Vehicle (PV) to test if the distance is safe or not: PV is the *SceneObject*, Obstacle is an *ObjectType*, and TTC is a *ParameterType* created in PVParametersLibrary, a parameter library created for PV. TTC can be measured in seconds (*PrimitiveParameterType float*).

PropertyTypeLibrary contains properties. Each property contains states representing the situations the object faces to assess its safety in the scenario. Let us suppose that EgoPropertiesLibrary is the library created for the Ego car containing properties such as a preceding_vehicle_distance property. This property returns the distance between the Ego and the Preceding Vehicle (PV). The states of this property can be a safe_distance which validates that the distance between

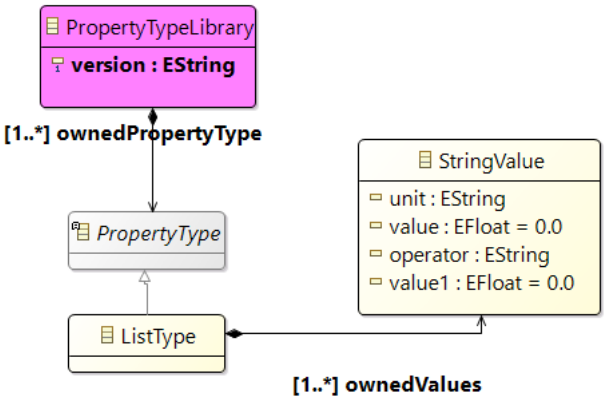

Figure 3.11: *PropertyTypeLibrary* containing properties composed of states that can have values.

EV and PV is safe, and an emergency_distance which confirms it is an unsafe distance. The property can not be assigned to more than a state at a time slot: the preceding_vehicle_distance is either a safe_distance or emergency_distance and not both. Both states are assigned to *StringValue* types. The safe_distance can be 2 seconds (operator: =, value: 2, unit: seconds) [90].

The *ParameterTypeLibrary* always depend on the *ObjectType*. This means in case we delete the *ObjectType*, we will lose the related parameter library. This is where came the idea of versioning what remains in case of loss of an *ObjectType*. We gave a *version* string to the *PropertytypeLibrary* same as for *ObjectTypeLibrary*. In case of any update, we give new versions to these libraries. This is how rules cover conditions that consist of testing properties or parameter values for each *objectType*.

3.3.2.3 Expression and SelectByGoal domain concepts

As mentioned previously, an *Expression* is referred to an *alert* and an *action* as shown in Figure 3.12. We create a new representation model and hide all the other elements just to represent the *Expression* domain concept. This is the reason why *alert* and *action* are shown as attributes with an association logo. The main expressiveness of our rules is based on Boolean invariants. So we use classical logical operators to build expressions as seen in Figure 3.12. An *Expression* can contain logical operators and a *TestValue* that is a *PropertyType* with its *StringValue* as seen in Figure 3.13. We use the following logic operators defined from [91]:

- The logical *AND* (&&) operator (logical conjunction) for a set of Boolean operands is true if and only if all the operands are true. Otherwise, it is false. AND operator should at least contain two expressions (which explains

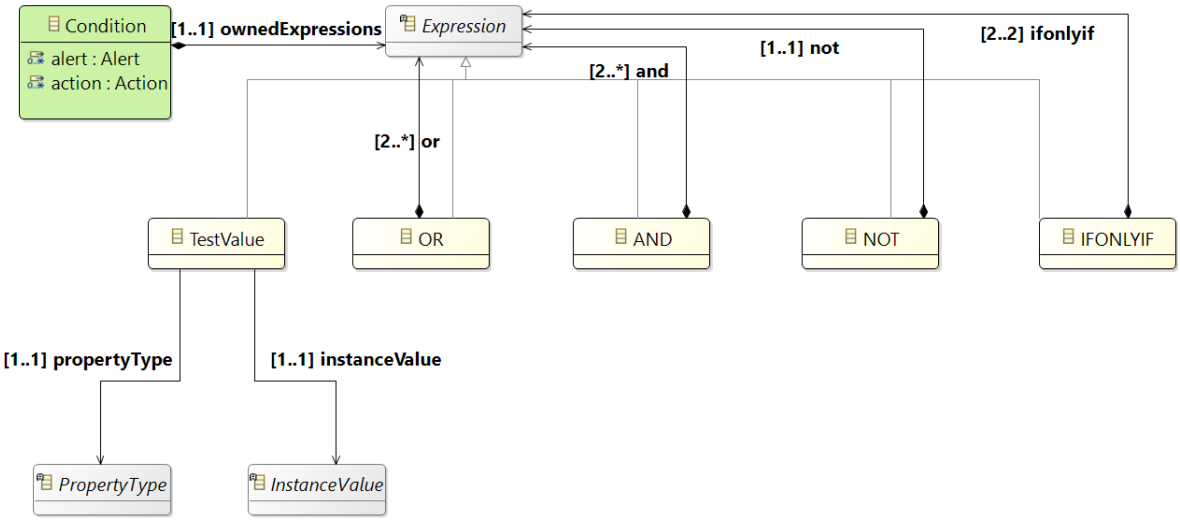

Figure 3.12: Expression abstract metamodel using logical operators.

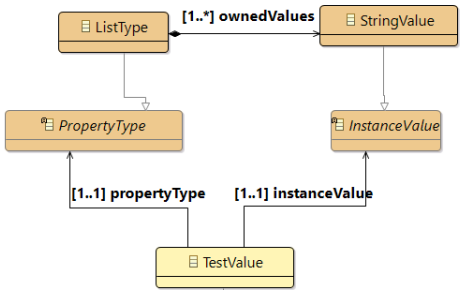

Figure 3.13: *TestValue* referring an *InstanceValue* that is a *StringValue* to its *PropertyType*.

the cardinality of [2..*])

- The logical *OR* (||) operator (logical disjunction) for a set of operands is true if and only if one or more of its operands is true. It is typically used with Boolean (logical) values. When it is, it returns a Boolean value. However, the || operator actually returns the value of one of the specified operands, so if this operator is used with non-Boolean values, it will return a non-Boolean value. OR operator should at least contain two expressions (which explains the cardinality of [2..*])

- The logical *NOT* (!) operator (logical complement, negation) takes truth to falsity and vice versa. It is typically used with Boolean (logical) values. When used with non-Boolean values, it returns false if its single operand can be converted to true; otherwise, it returns true. NOT operator should contain one expression (which explains the cardinality of [1..1])

- The *IFONLYIF* operator that acts as an equivalence operator. If the operands are equals, *IFONLYIF* returns true. It returns false otherwise. It is the equivalence operator described in 5.7. *IFONLYIF* operator should only contain two expressions (which explains the cardinality of [2..2])

A *Goal* can contain at least one *Condition*. Each *Condition* is composed of one *Expression* as seen in Figures 3.8 and 3.12.

It is also important to prioritize rules in case of a behavior's contradiction. We show in Figure 3.14, two rules (or Goals). The first rule leads to a slight

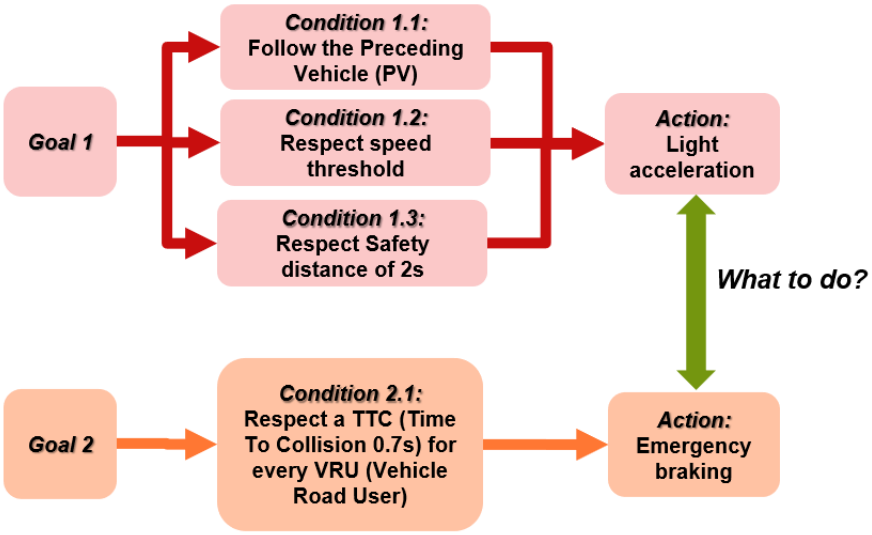

Figure 3.14: Case where emergency braking should have a higher priority than light acceleration.

acceleration. The second rule leads to execute an emergency braking. These two actions are contradictory and can not be triggered at the same time. Goal 1 consists of having three conditions : (1) following the Preceding Vehicle (PV), (2) respecting the speed threshold, and (3) respecting the safety distance of 2 seconds. If the three conditions are met together at the same time, the first goal leads to a light acceleration, contrary to the second goal which generates emergency braking. The second Goal consists of respecting a Time To Collision (TTC) for every Vehicle Road User (VRU), e.g. pedestrians.
Imagine that the engineer defines these two goals in a conjonctive way. We can have a case where all the conditions are met and both actions are triggered at the same time. In this case, emergency braking should have a stronger priority than light acceleration, and this priority needs to be carefully defined within the execution type sorting.

This is why we introduce the notion of *SelectByGoal* in Figure 3.15 that allows

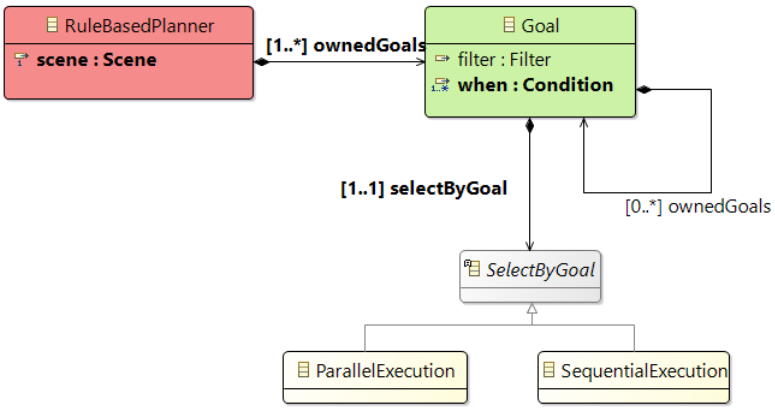

Figure 3.15: SelectByGoal notion to categorize rule execution.

the user to execute their goals with their conditions in parallel (*ParallelExecution*) or sequentially (*PriorityExecution*). We sum up the *ParallelExecution* with *Constraint* goal type, and *PriorityExecution* with *Priority* goal type.
Note that the *Goal* domain concept is the same as the *Goal* introduced in Figure 3.8, but due to the creation of a new representation model, the associations became attributes preceding to a logo.
If a goal has a priority goal type, the second is not executed unless the first is false. If a goal has a constraint goal type, we activate a parallel execution. The same concept is applied to the conditions for each goal. A goal can be made up of several conditions (when..then). If this goal is given a priority goal type, all conditions will execute sequentially. If we give this goal a constraint goal type, all the conditions will run in parallel. Let us take a more complex example of having two goals: Goal1{Condition1, Condition2} and Goal2{Condition3, Condition4}. We can have four cases to the *SelectByGoal*, which is for us a goal type, as follows:

1. If Goal1 and Goal2 have a priority goal type, then the execution starts with Condition1. If Condition1 is false, then the system executes Condition2. If Condition2 is false, then the system executes Condition3. If Condition3 is false, then the system executes Condition4.

2. If Goal1 has a priority goal type, and Goal2 has a parallel constraint type, then the execution starts with Condition1, then Condition2. If Condition2 is false, then the system executes in parallel Condition3 and Condition4.

3. If Goal1 has a parallel constraint type, and Goal2 has a priority goal type, then the execution starts with Condition1 and Condition2 and Condition 3 in parallel. If Condition3 is false, then the system executes Condition4.

4. If Goal1 and Goal2 have a parallel constraint type, then the execution starts with Condition1, Condition2, Condition3, and Condition4 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. If the engineers want to give priority to a rule, they use the Priority goal type, otherwise the choose the Constraint goal type.

3.3.2.4 AlertLibrary and ActionLibrary domain concepts

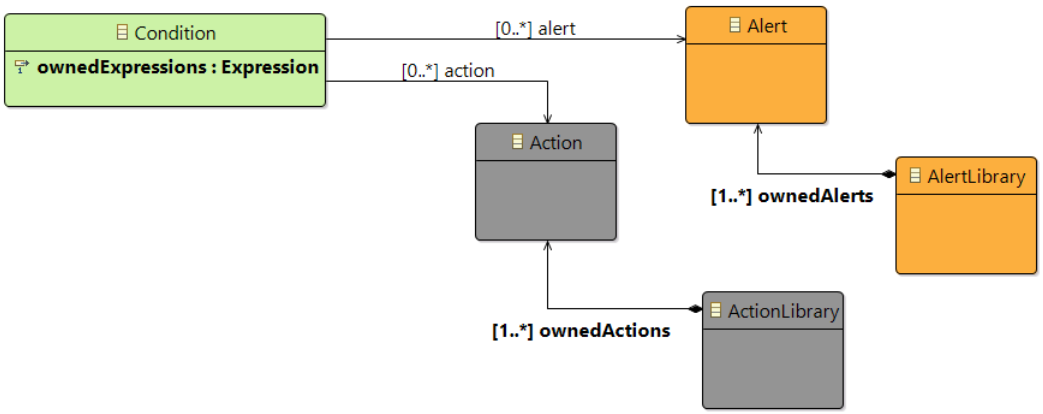

Figure 3.16: Libraries of actions and alerts.

As mentioned earlier, each *Goal*, having a *SelectByGoal* to specify the type of execution, is composed of at least one *Condition* containing an *Expression* using logical operators. The metamodel can also filter the rules either by role or by expression. Each *Condition* shall trigger a behavior in case of an occuring violation. This is why, it refers to alerts and actions that represent the behavior to be triggered.

We created libraries for the actions and alerts displayed in Figure 3.16. Actions that can be integrated into the motion planner. Alerts have an informative objective to alert the user of the situation. Creating this metamodel gives us the right to a more concrete description of the environment and the rules described in 3.3.3.

3.3.3 Concrete syntax

While metamodeling is now well understood for the definition of abstract syntax, we explain in this Section the definition of concrete syntax. Concrete syntax definition is an important part of metamodeling [49]. We choose to have a textual concrete syntax to match the classical paper-papers requirements used in Renault's process. The semantics is given by transformation [92] to C code in a first step and to propositional logic expressions in a second step. This Section

describes the concrete syntax. Figure 3.7 shows the integration of the concrete syntax within the Gemoc framework. We use Xtext technology to provide a concrete textual syntax to our language.

We create Xtext files for the main domain concepts presented and described in our metamodel in the previous Section 3.3.2. These are presented in Figure 3.17. It grants grammar description of the *goals* and their priorities in Section 3.3.2.3, the *scene* in Section 3.3.2.1 with the *parameters* and *properties* in Section 3.3.2.2, and *actions* and *alerts* in Section 3.3.2.4. We have file extensions for these domain concepts in a working environment, so the programming environment identifies the grammar to use. The extensions defined for each domain concept serve the user to create new files and libraries formally.

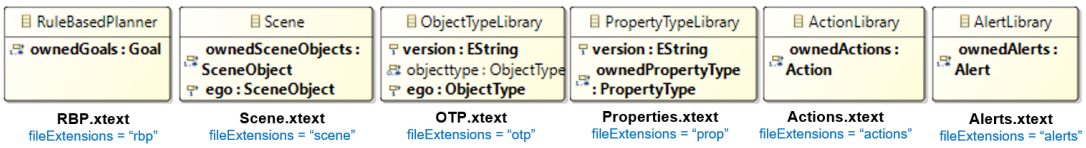

Figure 3.17: Concrete Xtext Files referred to domain concepts in our metamodel.

With Xtext we define the grammar for the language. As a result, we get for each object (i.e. domain concept) Xtext project with full infrastructure. We detail the grammar for each object in the following subSections and link them to the instance models which we present later.

3.3.3.1 RuleBasedPlanner grammar

RBP file (with *.rbp* extension) is central for the user to define his safety rules and depends on other models which are introduced later. For the *RuleBasedPlanner* concept, we present a part of its grammar we used in Figure 3.18. An instance model of the rule-based planner grammar is presented in Section 3.3.4. We also present a detailed use case in Chapter 6 in Figures 6.14, 6.15, 6.16, 6.17, and 6.18.

The *RuleBasedPlanner* possesses a *name* that has a *EString* type. Seen that it also refers to a *Scene*, *scene=[Scene—EString]* means that another file (which has *.scene* extension) contains all data for the scene (detailed in Section 3.3.3.2). The cardinality of *(ownedGoals+ =Goal)+* in the *RuleBasedPlanner* means that the user should at least define one *Goal*, as we can see in Figure 3.15. The cardinality of *(ownedGoals+ =Goal)∗* in the *Goal* means that the user is not forced to define a *Goal* in a previously defined one. Each *Goal* introduces one *Condition* (*WHEN*) at least that constitutes logics expressivity, since the + cardinality refers to [1..*]. Each *Condition* can refer to *Action* and *Alert* domain concepts since the ∗ cardinality refers to [0..*]. Since it is a reference for the alerts and actions, this means that other files (which have *.alerts* and *.actions* extensions)

```
RuleBasedPlanner returns RuleBasedPlanner:
    'RuleBasedPlanner' name=EString
    '{'
        'scene' scene=[Scene|EString]
        (ownedGoals+=Goal)+
    '}';

Goal returns Goal:
    'GOAL' name=EString
    '{'
        ('GoalType' selectByGoal=SelectByGoal)
        ('filter' filter=Filter)?
        (when+=WHEN)+
        (ownedGoals+=Goal)*
    '}';

WHEN returns Condition:
    'WHEN'
    '{'
        ownedExpressions=Expression
        'action' (action+=[Action|EString])*
        'alert' (alert+=[Alert|EString])*
    '}';
```

Figure 3.18: RBP.xtext project for the textual representation of RuleBasedPlanner description.

contain all data for the behaviors (detailed in Section 3.3.3.5). Each *Goal* has a goal execution type (*Constraint* or *Priority*) as seen in Figure 3.19, and can be filtered either by *Role* or by *Expression*.

```
Filter returns Filter:
    SelectByRole | SelectByExpression;

SelectByGoal returns SelectByGoal:
    SequentialExecution | ParallelExecution;

SequentialExecution returns SequentialExecution:
    name="Priority";

ParallelExecution returns ParallelExecution:
    name="Constraint";

SelectByRole returns SelectByRole:
    'Role' role=[ObjectType|EString]
    ;

SelectByExpression returns SelectByExpression:
    'Expression' expression=Expression
    ;
```

Figure 3.19: Textual representation of Goal features in the RuleBasedPlanner description.

3.3.3.2 Scene grammar

Scene file (with *.scene* extension) serves for the user to define the objects seen in the scenarios and their types. It captures the perception capabilities of the

autonomous vehicle. For the *Scene* concept defined in Section 3.3.2.1, we present a part of its grammar in Figure 3.20. The *Scene* owns a *name*, refers to the *ego*

```
Scene returns Scene:
    'scene' name=EString
    '{'
        'ego' ego=[SceneObject|EString]
        (ownedSceneObjects+=Ego)
        (ownedSceneObjects+=SceneObject)+
    '}';

SceneObject returns SceneObject:
    'role' name=EString 'objectType' objectType=[ObjectType|EString]
    ;

Ego returns SceneObject:
    'role' name="Ego" 'objectType' objectType=[ObjectType|EString]
    ;
```

Figure 3.20: Scene.xtext project for the textual representation of Scene description.

car, and is composed of at least one *ownedSceneObject* other than the *ego*. Every *ownedSceneObject* has an *ObjectType* that is detailed below. The user defines the concrete files using *.scene* extension in the application created outside the workspace. An instance model of the scene's grammar is presented in Section 3.3.4. We also detail a use case in Chapter 6 in Figure 6.11.

3.3.3.3 ObjectTypeLibrary grammar

OTP file (with *.otp* extension) serves for the user to define the ego and the types' objects. It also serves to specify what are the parameters used to assess safety, and to refer each *ObjectType* to a *PropertyTypeLibrary*. For the *ObjectTypeLibrary* presented in Section 3.3.2.2, we present a part of its grammar in Figure 3.21. The *ObjectTypeLibrary* owns a *name* and a *version*. It is composed of one type for the *ego*, and can have other types for the *ownedSceneObject*. Every *ObjectType* can have a *PropertyTypeLibrary* and another *parameterTypeLibrary*. As for the *ego* type, it has to contain one ego property library, that is *EgoObjectType*, to make the expressions valid. An instance model of the object type library grammar is presented in Section 3.3.4. We also present a detailed use case in Chapter 6 in Figure 6.10.

3.3.3.4 PropertyTypeLibrary grammar

Properties file (with *.prop* extension) serves for the user to define all the properties and states. For the *PropertyTypeLibrary* presented in Section 3.3.2.2, we present a part of its grammar in Figure 3.23. Every *PropertyTypeLibrary* has a *name* and a *version*, and contains a list of properties, or states, *(ownedPropertyType+=ListType)+*. Each property can have different values *(owned-*

```
ObjectTypeLibrary returns ObjectTypeLibrary:
    {ObjectTypeLibrary}
    'ObjectTypeLibrary' name=EString 'version' version=EString ':'

        ego=EgoObjectType
        (objecttype+=ObjectType)+
    ;

ObjectType returns ObjectType:
    name=EString'{'
        ('PropertyLibrary'propertyLibrary=[PropertyTypeLibrary|EString]";")?
        'ParameterLibrary' parameterLibrary=ParameterTypeLibrary
    '}';

EgoObjectType returns ObjectType:
    name='Ego''{'
        'EgoPropertyLibrary' propertyLibrary=[PropertyTypeLibrary|EString]';'
        'EgoParameterLibrary' parameterLibrary=EgoParameterTypeLibrary
        '}';
```

Figure 3.21: OTP.xtext project for the textual representation of ObjectTypeLibrary description.

```
ParameterTypeLibrary returns ParameterTypeLibrary:
    name=EString ':'
        (ownedParameterType+=PrimitiveParameterType)+
    ;

EgoParameterTypeLibrary returns ParameterTypeLibrary:
    name='Ego' ':'
        (ownedParameterType+=PrimitiveParameterType)+
    ;

PrimitiveParameterType returns PrimitiveParameterType:
    'parameter' name=EString 'Type' kind=PrimitiveTypeKind';'
    ;

enum PrimitiveTypeKind returns PrimitiveTypeKind:
    int32 = 'int32' | float = 'float' | boolean = 'boolean' | string = 'string' | int64 = 'int64';
```

Figure 3.22: ParameterTypeLibrary textual representation in ObjectTypeLibrary description.

Values+=StringValue)+. Each value can be constant by giving it a specific value with its unit and operator, or can be a variable. We can see an instance model of this grammar in Section 3.3.4. We detail other instance models in Chapter 4 in Figures 4.6 and 4.7, and in Chapter 6 in Figure 6.12.

3.3.3.5 AlertLibrary and ActionLibrary grammars

Alerts and actions files (with the extensions *.alerts* and *.actions*) allow the user to define the necessary alerts and actions. For the actions and alerts libraries presented in Section 3.3.2.4, we present their grammar in Figure 3.24. Each *Alert* and *Action* has a *name* with *EString* type. An instance model of the alert and action libraries' grammar is presented in Section 3.3.4. We also present another examples in Chapter 6 in Figures 6.8 and 6.9.

```
PropertyTypeLibrary returns PropertyTypeLibrary:
    'PropertyTypeLibrary'
    name=EString
    '{'
        'version' version=EString
        (ownedPropertyType+=ListType)+
    '}';

ListType returns ListType:
    'state' name=EString 'CanBe' (ownedValues+=StringValue)+
    ;

StringValue returns StringValue:
    {StringValue}
    name=EString ('operator' operator=EString)?
    ('value' value=EFloat)?(value1=EFloat)? ('unit' unit=EString)?;
```

Figure 3.23: Properties.xtext project for the textual representation of PropertyTypeLibrary description.

```
AlertLibrary returns AlertLibrary:
    'AlertLibrary' name=EString ':'
    (ownedAlerts+=Alert)+
    ;

EString returns ecore::EString:
    STRING | ID;

Alert returns Alert:
    {Alert}
    'Alert' name=EString';';
```

(a) Alerts.xtext definition.

```
ActionLibrary returns ActionLibrary:
    'ActionLibrary' name=EString ':'
    (ownedActions+=Action)+
    ;

EString returns ecore::EString:
    STRING | ID;

Action returns Action:
    {Action}
    'Action' name=EString';';
```

(b) Actions.xtext definition.

Figure 3.24: Grammar for Alert and action libraries.

3.3.4 RBP and libraries illustration using EPSAAV specifier

This Section illustrates the abstract and concrete syntaxes defined previously. We represent an example of files instantiated. This is considered the interactive console that the user will use to define his libraries and requirements. As mentioned previously, the user have to create six files as following:

- A file for the rule-based planner named *RBP.rbp* containing all the rules with the assigned behaviors as seen in Figure 3.25. It is based on the grammar defined in Section 3.3.3.1. We can see the Scene assignement and two sequential rules: the first deceleration Goal serves of alerting the driver to decelerate when the front_car_distance is unsafe. This rule has a priority over the acceleration Goal if the distance between Ego and PV is safe. Since the second rule does not contain more than one condition and does not succeed any rule, it does not matter if it has a constraint or prioirty goal type.

```
RuleBasedPlanner RBP{
    scene ^Scene
    GOAL deceleration{ //unsafe PV distance
    GoalType Priority
    WHEN{
        propertyType "EgoProperties.front_car_distance" is
        "EgoProperties.front_car_distance.unsafe_distance"

        action "Actions.decelerate"
        alert "Alerts.longitudinal_deceleration"
    }
        GOAL acceleration{ //safe PV distance
            GoalType Constraint
            WHEN{
                propertyType "EgoProperties.front_car_distance" is
                "EgoProperties.front_car_distance.safe_distance"

        action "Actions.accelerate"
        alert "Alerts.longitudinal_acceleration"
            }
        }
    }
}
```

Figure 3.25: RBP illustration.

- A file for the scene named *Scene.scene* containing the objects perceived in the scenario as seen in Figure 3.26. It is based on the grammar defined in

```
scene Scene {
    ego ^Ego
    role Ego objectType "OTL.Ego"
    role "preceding_vehicle" objectType "OTL.Obstacle"
}
```

Figure 3.26: Scene illustration.

  Section 3.3.3.2. We chose only to perceive the Ego car (the AV) with an Ego object type, and the PV with an Obstacle object type.

- A library for the object types named *ObjectTypeLib.otp* as seen in Figure 3.27. It is based on the grammar defined in Section 3.3.3.3. For the Ego

```
ObjectTypeLibrary OTL version'1':

Ego {
    EgoPropertyLibrary EgoProperties;
    EgoParameterLibrary Ego :
        parameter longitudinal_Acceleration Type float;
}

"Obstacle"{
    ParameterLibrary Obstacle :
        parameter id Type int32;
        parameter ttc_min Type float;
}
```

Figure 3.27: Object type library illustration.

  type, we need the longitudinal_acceleration parameter, and for the Obstacle type, we need the ttc_min and the id with float and integer units respectively.

- A library for the Ego properties named *EgoProperties.prop* as seen in Figure 3.28. It is based on the grammar defined in Section 3.3.3.4. It contains one

```
PropertyTypeLibrary EgoProperties
{   version '1'
    state "front_car_distance" CanBe
    "safe_distance" operator">=" value 2.0 unit "s"
    "unsafe_distance" operator"<" value 2.0 unit "s"
}
```

Figure 3.28: Ego properties illustration.

property to check the distance with the PV. We named it front_car_distance, and have two states: safe_distance (if it is above or equal to 2 seconds) and unsafe_distance if it is less than 2 seconds).

- A library for the alerts that we named *Alerts.alerts* as seen in Figure 3.29. It is based on the grammar defined in Section 3.3.3.5. It contains two alerts

```
//this is the definition of the library for the alerts

AlertLibrary Alerts :
Alert "longitudinal_acceleration";
Alert "longitudinal_deceleration";
```

Figure 3.29: Alert library illustration.

triggered to the driver: the longitudinal deceleration and acceleration.

- A library for the actions that we named *Actions.actions* as seen in Figure 3.30. It is based on the grammar defined in Section 3.3.3.5. It contains two

```
//this is the definition of the library for the actions

ActionLibrary Actions :
Action "accelerate";
Action "decelerate";
```

Figure 3.30: Action library illustration.

actions triggered to the driver: decelerate and accelerate.

3.3.5 Generation of artefacts

We explain in this Section the generation of artefacts relying on what is described in Section 2.7.3. This step is important as the semantics is given by transformation. To generate artefacts, we use the instance model created by the safety engineer. These instance models are created by using the abstract and concrete model which we described before in Sections 3.3.2 and 3.3.3 as seen in Figure 3.7.

We use the **EPSAAV generator** to achieve code and text generation from models while making no restrictions on the visual appearance of the metamodel.

We use Xtend as a support for code and text generation. It supports auto-indentation and is fully integrated with the Eclipse IDE [51]. Code and text generations do not cross the language boundaries. We use what is included in the previous syntaxes for the generation. We proceed to three different types of generation as seen in Figures 3.31 and 3.32 and described further. The code

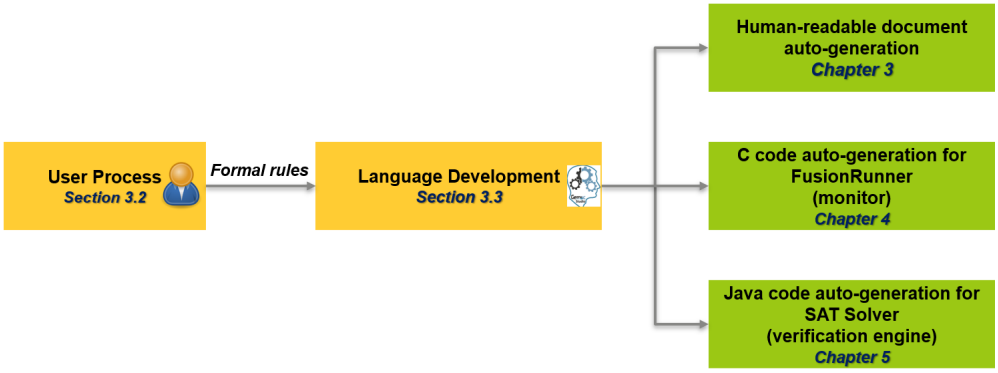

Figure 3.31: Three types of generated artefacts from the EPSAAV safety rules.

```
    //We generate a documentary text file that explains what is described by the engineer

    fsa.generateFile('Document.txt', res_text)

    //We generate also a C code for the goals, properties and parameters, scene, alerts and actions.

    fsa.generateFile('...c', res_code)
    fsa.generateFile('...h', res_header)

    //for the RBP we generate also a .java code to the SAT solver

    fsa.generateFile('sat4j_rules.java', res_java)
}
```

Figure 3.32: Artefacts coded to be generated in Xtend project.

generators created in this thesis are intended for C and Java languages but could be adapted to other languages required in the whole design flow.

3.3.5.1 Human-readable document generation

We generate for each user environment definition a human-readable document that describes all the input data that he introduced. Figure 3.33 presents the document generated from the illustrations presented previously in Section 3.3.4. Another application of an instance model for a text generation and described in Chapter 6 in Section 6.5 and added to the Appendix A.2. This helps him re-evaluate his decision, and communicate with engineers to improve the system development.

```
BEGIN GOAL deceleration{
    Goal type: Priority
    when{
            front_car_distance is unsafe_distance
    }
    then behavior_deceleration_cond1:
        executing decelerate
        longitudinal_deceleration

    BEGIN GOAL acceleration{
        Goal type: Constraint
        when{
                front_car_distance is safe_distance
        }
        then behavior_acceleration_cond1:
            executing accelerate
            longitudinal_acceleration

    }
}
BEGIN RESOURCES
    resource longitudinal_acceleration
    resource longitudinal_deceleration
END RESOURCES

BEGIN ACTIONS
            action accelerate
            action decelerate
        END ACTIONS

BEGIN STATE VECTOR
PropertyTypeLibrary EgoProperties{
        state front_car_distance CanBe  safe_distance  unsafe_distance
}
END STATE VECTOR
```

Figure 3.33: Example of a human-readable document generated.

The human-readable generation for this thesis is a text document but can be adapted to another format. This document contains all libraries and requirements. It can be considered as a report that can be used as a reference at the company level to disseminate the safety policy to be applied in the company. It is useful and time-saving to generate a human-readable document. This document was at the center of the process. Our major contribution is show that a model-based approach allows for building a mode trustable process while still being able to produce the exact same document as a conservative measure to remain compatible with the initial process.

The first document generation is long to set up and it requires a lot of effort to get familiar with the document generation approach and the associated tools. After a few iterations, as the system definition and specifications evolve, the document is changed and needs to be updated. In the short term, creating a document (architecture, interface, specification, verification plan) is always faster by hand the first time. But once the method generation of the document is set, it is automatic and changes can be applied faster in a systematic way.

Safety experts nowadays use handwritten techniques instead of generated documents. They may realize that it becomes tedious to identify the parts of the

document that must be updated and to perform those modifications. In this case, the model and the document share the same information and face changes and risks of inconsistencies. The engineers spend most of the effort on synchronization instead of engineering. This is how document generation brings value and constitutes the engineers reference instead of the model.

3.3.5.2 Monitor generation

Most safety experts tend to use hand-written English documents to capture safety requirements. Pure textual documents are prone to errors as there is no way to actually detect flaws.

This is why we introduced this proposal to help him avoid wasting time in development and reduce code errors. To do so, we generate a code, which we called monitor system, compatible with the simulator used. In our case, we produce C code that is compatible with the *FusionRunner* debugger used for simulation. We elaborate further on this in Chapter 4. The application of an instance model is detailed in Chapter 6 in Section 6.6.

3.3.5.3 Verification rules generation

We also want to prevent inconsistencies in the rules defined. To achieve this, we generate another code, that we call verification rules. Those rules are a translation of safety rules into first-order predicates. We can then use a satisfiability analyses to detect inconsistencies that may arise within the safety requirements. In our case, we produce a Java code described in Chapter 5 that is relies a SAT solver used in big industrial problems. The application of an instance model is detailed in Chapter 6 in Section 6.7.

3.4 Conclusion

In this Chapter, we have presented our EPSAAV framework and how the language process development is implemented using Gemoc Studio: an abstract syntax that defines the EPSAAV metamodel and frames the source of truth, a concrete syntax that enables engineers to define rules and libraries using **EPSAAV specifier** in an interactive console, and an operational semantic for artefacts generation. We addressed a case study in which we specify an instantiation model to evaluate the feasibility of EPSAAV language. The user instantiates the model by creating six initial files with specific extensions; RBP (*.rbp*), alerts and actions libraries (*.alerts* and *.actions*), object type library that contains parameters and properties (*.otp* and *.prop*), and scene (*.scene*) that can be used as a console for testing or validation purposes.
We mentioned the real use of generating text documents in helping users compare their alternatives and saving time for synchronization. Each type of code

generated corresponds to a specific step in the design and validation process. One monitor generated in C language is compatible with a simulator to detect ambiguities when violations occur while executing various scenarios. We detail the monitor and the *FusionRunner* debugger in the following Chapter 4. A set of verification rules are generated in a language compatible with a industrial-strengh SAT solver to study inconsistencies in rules. To do this effectively, we address the study in Chapter 5.

Chapter 4

Generation of a Monitor for Renault Simulation Environment

4.1 Introduction

The automotive industry has started working on improving the safety field. Self-driving cars can potentially identify hazards better than people in several circumstances. They combine sensors and software to control, navigate, and drive the vehicle. This helps avoid human errors due to inattention or tiredness, as AVs may have good perception capabilities and be less vulnerable to incapacitation, such as distraction or fatigue. To do this, safety rules must be explicitly captured and integrated together with traditional functional requirements. Safety assessments of the AVs came as an important aspect in the development of safe autonomous systems [93, 94, 95]. ISO 26262 defines functional safety for automotive equipment applicable throughout the lifecycle of all automotive electronic and electrical (E/E) safety-related systems. This standard only deals with malfunctions that may arise from E/E systems. Therefore, the traditional methods to assess the safety and evaluate driver assistance systems are no longer sufficient to reach higher-level of autonomy in AVs [28]. They are not feasible to complete the huge amount of testing required by these methodologies [96]. To not postpone the progress of AVs, it is crucial to develop new assessment methods [93] that take into consideration the environmental conditions under which AVs evolve. We presented in Chapter 2 Section 2.2 an overview of safety approaches. Drawbacks are mentioned in *Data Driven Safety* and *Disengagements* approaches as they require very large amount of data that would take hundreds of years to gather so as to ensure safety. This is problematic. *Simulation* is a solution to deal with any day-to-day situation, which means any time of the year, any weather or traffic conditions, and every road in the world. Sometimes, a slight difference between the simulation and reality can change the results. Addressing a tool that includes both simulated and real data is an important solution to support

both approaches. Another challenge is the interaction between the driver and the system. By building a sophisticated autonomous system, we can send alerts to the driver and the safety engineer to interact and identify causes of dangerous situations.
Therefore, in this thesis, we present our *Scenario-based Verification* approach, in which we aim to expose the AV to scenarios known as dangerous situations by using a debugger to trigger safety notifications and study the reaction of AV exposed to complex driving tasks The objective of this Chapter is to present the simulation environment for evaluating the safety and to explain the choice of using *FusionRunner* as a resimulation tool at Renault. In doing so, we support the SOTIF standard that advocates for scenario-based test methods.
This Chapter is organized as follows: Section 4.2 introduces the sensors' data challenges and how we can integrate them into the EPSAAV language. In Section 4.3, we represent the Autonomous Driving (AD) software architecture where the simulation environment plays its role, and describe the *FusionRunner*. The proposed methodology to verify safety using *FusionRunner* complies to the SOTIF standard by adopting an assessment based on traffic scenarios. To analyze and verify traffic scenarios on real-world driving data, we introduce in Section 4.4 our process to generate monitors. We show how we connect these monitors to the *FusionRunner* in Section 4.5. Finally, Section 4.6 summarizes this entire Chapter.

4.2 Sensing data for AV

SOTIF considers the environmental conditions that lead to unsafe and inappropriate situations. It is important to know the source of the failures that are independant from the AV system. Sensing data is an important task to verify and check safe and unsafe cases, and track sensors availability to evaluate misuses treated by SOTIF, e.g loss of communication. All driving tasks can be broken down into three components (see Figure 4.1). By understanding what is happen-

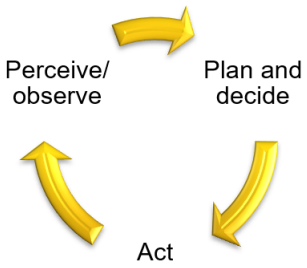

Figure 4.1: Tasks in an autonomous system.

ing around the ego car, one can perceive its surroundings and analyze ego motion and the environment. After that, the system needs to make a driving decision,

such as accelerating or braking (e.g., when AV perceives a pedestrian entering the roadway) [97].

4.2.1 Perceiving the environment sensors

A reliable perception system must be able to extract information from the external environment for proper planning and decision-making. Perception systems are used to detect vehicles, humans, and other objects around the vehicle using sensors. AVs use combinations of exteroceptive and proprioceptive sensors and technologies to detect roadways, lanes and lines, and other static and dynamic objects.

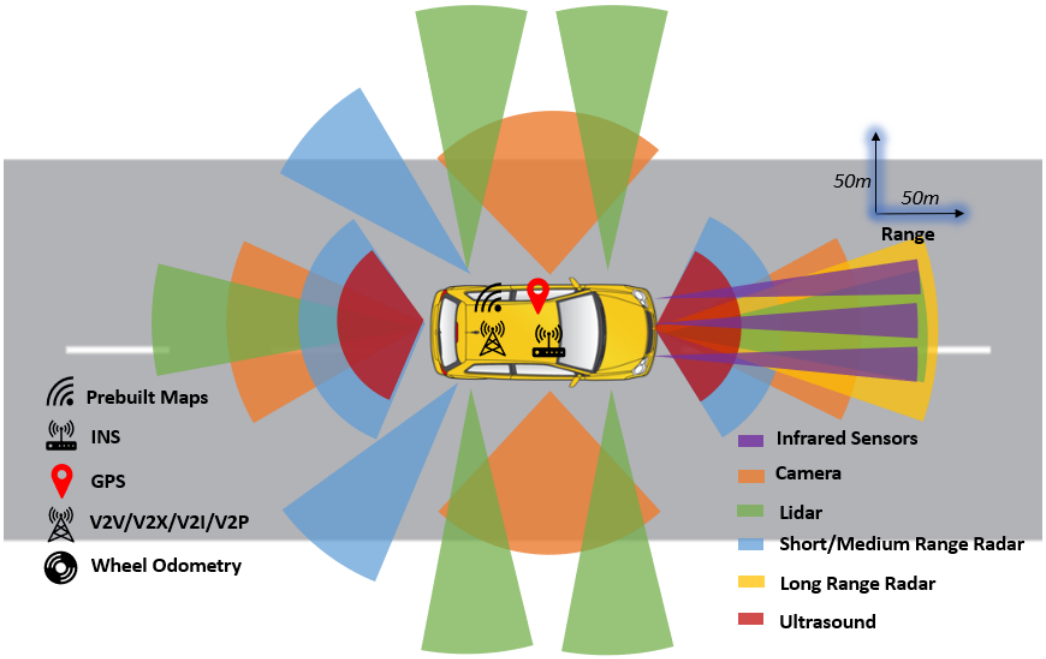

Figure 4.2: An example of sensor set for AV.

Figure 4.2 shows an example of a sensor set for AV along with the type and detection zones of each sensor. For instance, cameras can give a panoramic view, a digital side mirror if they are on the sides, traffic sign recognition, lane departure warning, etc. Sensors can be used like the Yaw Rate Sensor and the driver's camera which could be helpful for safety monitoring in case of a takeover control and other sensors that could be useful in terms of increasing the reliability of the system. Any driving task requires the Object and Event Detection and Response (OEDR) [98] to identify objects around the AV, recognize events occurring nearby, and then respond to them. For any agent or element on the road, the AV must identify what it is and understand its motion. There are two crucial parts to perception:

1. static objects that segregate regions on-road or even off-road regions such as the road, lane markings, traffic lights, and signs,

2. dynamic objects that are crucial for making informed driving decisions such as reactions to surrounding vehicles and pedestrians, and ego movement or localization, which are a key point for a robust perception and trustworthy decisions.

We need to be able to estimate where the Ego Vehicle (EV) is and how it is moving at all times by knowing the position to inform the system and make safe driving decisions. The data used for ego movement estimation can come from Global Positioning System (GPS), Inertial Measurement Unit (IMU), odometry sensors, and other predefined maps to strengthen the system. Depending on the incoming source, data are more or less reliable and may be more subject to errors. The main problem is that the sensor measurements may not be precise, and the precision does not provide information about the general error. This depends on various factors, including the conditions of use of the sensors. These factors do not have geometric properties.
Safety assessment can be done considering either raw sensor data or fused data which combines several raw sensor data. In this thesis, we choose to assess safety by relying on the fused data. To check whether the AV is in a safe situation or not, we need a module that contains all the variables, parameters, and information regarding the safety of the ego car. Errors and persistence settings should also be specified to investigate safety scenarios. These arguments have an impact on safety and affect decision planning. We describe further what are the environmental condition challenges that the AV faces and may affect its safety. We give examples of errors and persistence and how we implement these requirements using the **EPSAAV specifier**.

4.2.2 Perception challenges

There are various approaches to perceiving the environment with the associated challenges to be faced.

4.2.2.1 Sensor uncertainty

Each sensor has its strengths and weaknesses in terms of range, detection capabilities, and reliability in different environments. Providing redundancy is necessary to safely sense the environment, but the more sensors AV used, the more complex algorithms are and the more expensive the car is. Sensor uncertainty is one of the challenges as it often happens that visibility becomes dazzling or measurements are corrupted [99]. For example, Lidar, Radar, and GPS measurements are noisy in terms of position values. An uncertainty on measurement should be

considered while addressing safety concerns. There is a crucial need to develop a system capable of addressing this problem.

4.2.2.2 Robust detection

Perception aims to have a robust detection and segmentation. Approaches with modern machine learning improve efficiency and accuracy and this is when the amounts of data increase [100, 101, 102]. Access to large datasets is critical to this effort, and it is a very expensive and time-consuming process. When studying safety, we need an exhaustive and comprehensive system that can catch errors and trigger alerts to the user.

4.2.2.3 Illumination lens flare

There are also effects such as drastic lighting changes and lens flare seen in Figure 4.3 and illustrated in [4], or GPS outages and tunnels that render some sensor data completely unusable or unavailable. Perception methods require multiple redundant information sources to overcome sensor data loss.

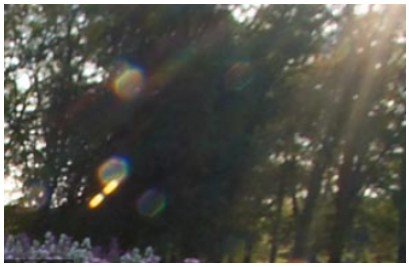

Figure 4.3: Example of lense flare illumination taken from [4].

4.2.2.4 Weather and precipitation

Weather conditions and precipitation can affect the quality of sensors' input data. Sometimes testing vehicles that do not use sensors immune to weather conditions can affect unforeseen input data efficiency as seen in Figure 4.4.

Figure 4.4: Weather in winter, rain and fog taken from [5].

4.2.3 Overcoming sensing challenges using EPSAAV language

33% of crashes only involved detection and perception factors [97]. Sensors most of the time have some degree of tolerance and exhibit errors. It is crucial to link this tolerance, the capacity of the sensors, and the degree of safety that can be ensured considering the confidence in the vehicle. Challenges can be described in error and persistence parameters which are important when evaluating safety.

The error parameter helps manage and bandage the error of all the sensors, for example in case of any failure or a breakdown. Each of the sensing technologies has limitations regarding sensing the environment in terms of supported range, accuracy, and resolution. Therefore, systems using these sensing technologies may have inconsistencies between the internal digital representation of the system environment and the actual environment, and thus there is a level of uncertainty in the actual state of the system, which has the potential to lead it to exhibit faulty behavior [103].

There is also the notion of continuity over time of objects that safety teams would like to be able to calibrate what the sensor detects and does not. The main problem is that the system is a bit disturbed and blinded with a probability of an occlusion. Sometimes perceiving an appearance and disappearance of the same object results in the inability of the system to determine whether the object exists or does not exist. The persistence parameter is affected by the confidence degree and plays an important role in these issues. In today's technologies, we have a vulnerability from this point of view, for example, if things are no longer visible or for example if we no longer see the lines, we must be able to bear this non-persistence.

```
state "stable_control" CanBe "stable" "not_stable"
```

Figure 4.5: Stable_control property to define perception stability.

```
state "front_car_distance" CanBe
"not_exist"
"safe_distance" operator">=" value 2.0 unit "s"
"acc_distance" operator">=" value 1.2 unit "s"
"strong_braking_distance" operator">=" value 0.8 unit "s"
"imminent_collision_distance"
```

Figure 4.6: Front_car_distance property containing states with different values.

Therefore, by using the EPSAAV language defined in Chapter 3, we give the user a facility to abstract away from some these challenges. These parameters and properties can be related to errors and persistence to help the user better

define safety requirements. The tool supports the inclusion of errors and persistence from sensors by defining new values and states in parameter and property libraries.

Figure 4.5 is an example of a property called **stable_control** following the grammar defined by the **EPSAAV specifier**, which represents a property to define AV's stability, for example, if there is something wrong with the sensors, then the control is not stable. Figure 4.6 is another example of a property called **front_car_distance** which illustrates how we define the different states for the front car distance and assign them values. For instance, it has a **safe_distance** state if the **front_car_distance** is equal or greater than 2 seconds. it has an **acc_distance** if it is equal or greater than 1.2 seconds, a **strong_braking_distance** if it is equal or greater than 0.8 seconds, and **imminent_collision_distance** when **front_car_distance** is less than 0.8 seconds. Using EPSAAV grammar, we can have states for each property to define its existence. Each type of perceived object can have its library of properties with different states. Figure 4.7 is another example using the same domain concept's grammar, which consists of adding a persistence value on detecting a line missing which illustrates how we define the different states when the car is not detecting or detecting the lines after. The threshold used in this case is 0.3 seconds but can be adapted depending on manufacturer internal choices. Us-

```
state "line_detection" CanBe
"stable"
"no_detection_in_less_than_t6" operator"<" value 0.3 unit "s"
"no_detection_in_more_than_t6" operator">" value 0.3 unit "s"
```

Figure 4.7: Line_detection property containing states with a given threshold.

ing these states we can measure line detection time, and a counter that returns **no_detection_in_less_than_t6**=1 when there is no line detection in less than 0.3 seconds, and **no_detection_in_more_than_t6**=1 if more than 0.3 seconds. In this way, the safety expert can use the tool to integrate all the parameters of perception errors and persistence to assess safety.

4.3 Representative AD software architecture

The information provided by the hardware components helps us represent the software architecture necessary to conduct the tasks. Figure 4.8 shows the AD's software architecture. As mentioned earlier, the environment is detected using fusion or raw data. These measurements (i.e. *Sensor Outputs*) are transmitted to the *Mapping* and *Perception* modules which are responsible for locating the AV and identifying elements of the environment such as bicycles, the road, road markings. The *Motion Planning* module decides actions and creates a path exit that should be both safe and comfortable for the driver. The execution of this plan

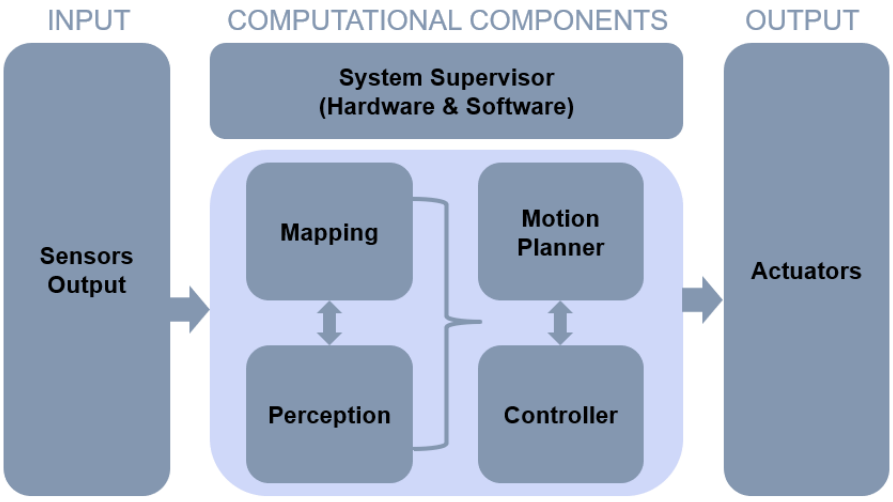

Figure 4.8: Representative software architecture of AD.

is done in the *Controller* module. This module is responsible for braking, steering, and positioning the vehicle to follow the planned path. It calculates current errors and tracks the local plan performance and adjusts current actuation commands to minimize errors in the future. The *System Supervisor* monitors all parts of the *Software* stack as well as the *Hardware* output. The *Software Supervisor* checks and ensures that all systems are working as intended, and if anything goes wrong, they are responsible for notifying the pilot of the problem found or a subsystem failure event. It also analyzes inconsistencies between the outputs of all modules. Simulators can be used to show the eventual real effects of each component and act as an alternative to replace the operation of a real process. They remain the convenient way to analyze how self-driving vehicles' perform over billions of miles of test drives, designed to keep them safe.

Simulator plays a key role in the AD software architecture. For instance, they can override the perception component that identifies elements and modules in the environment such as lines, ego, and objects as seen in Figure 4.9. They can also generate different scenarios and substitute the motion planner and the controller by taking as input the planning decisions from the ADAS. They enable vehicle designers to virtually test these scenarios quickly and cost-effectively. This not only accelerates development but also reduces the need for physical road testing. For example, model-based development tools help designers create AV simulators that have proven to be much more robust, less error-prone, and safer on the road [104]. Because the simulation is performed in a virtual environment, it continues to be faster, less expensive and provides more insight into the underlying physics than physical prototyping. These tools can also reduce development time, code validation and verification costs. In this thesis, we need a simulator capable of replacing perception with localization of the environment, and simulating tra-

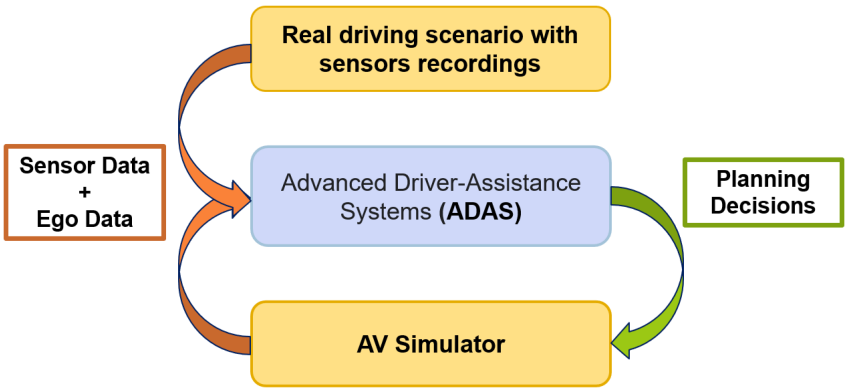

Figure 4.9: Real driving scenarios and simulators for ADAS.

jectory planning with useful safety information. EPSAAV language can use any existing simulator to evaluate safety. In this Section, we present the simulation environment used by Renault.

4.3.1 Advanced Driver-Assistance Systems (ADAS)

Advanced driver-assistance systems (ADAS) are groups of functionalities that help people in driving and parking tasks. ADAS increase vehicle and road safety by reducing road fatalities and limiting human errors. ADAS relies on sensors to perceive the environment and detect nearby obstacles or driver errors, and respond accordingly. ADAS follow the automation scale levels provided by SAE [35]. ADAS are among the fastest-growing segments in the automotive domain due to steadily increasing adoption of industry-wide quality and safety standards [105, 106].

Many features for ADAS have progressed to assist the driver in adopting functions. We can cite the following components that are important in the context of the thesis:

- Adaptive Cruise Control (ACC) can maintain a chosen velocity and distance between a vehicle and the vehicle ahead. ACC can automatically brake or accelerate with concern to the distance between both vehicles [107]. This system still requires an alert driver to take in his surrounding environment, as it only controls speed and the distance between the ego vehicle and the car in front.

- Autonomous Emergency Braking (AEB) is a system that automatically brakes if a collision with a target (vehicle, pedestrian, bicycle) is imminent. This is to avoid the collision or reduce the damage of the collision (if unavoidable) [108].

- Automatic Emergency Steering (AES) uses sensors and computer processing to detect when the ego vehicle could collide with an object in its path [109]. It applies automatic steering to mitigate or avoid the collision, even if the driver takes no action.

4.3.2 AV simulators

We can build, test, and debug embedded applications without a simulator [110]. However, there are several reasons why a simulator can make engineering tasks easier and save a lot of development time. Simulators do not replace an emulator, but the imitation of the operations of a real process. They are production and process planning support tools. They play an important role in ensuring that a successful system is designed in the shortest possible time.

We can find many benefits in using simulators. First, they save money and time while creating less expensive virtual experiences. Real assets take a lot of setup time for configuration and debugging. Second, they facilitate the verification and communication of ideas and concepts. Engineers gain confidence by visualizing and testing different conditions on the same or different scenarios without putting production at risk. Simulators are important in the field of safety to help safety engineers make the right decision before making changes in the real world. Third, they allow observing the behavior of the system over time at any level of detail. It also provides the experts with a realistic review unlike traditional techniques used to assess safety. Additionally, simulators can increase accuracy by capturing much more detail than an analytical test. It allows developers to virtually test millions of driving situations to assess safety on scenarios. It also helps manage uncertainty by allowing the quantification of risks while allowing for simple representations and more robust solutions. It is like a flight simulation where the pilot can make mistakes in a simulated environment and learn from his mistakes. System simulation can improve process understanding and minimize the risk of errors in real life. They provide sensor and ego data from the test simulations which are then transmitted to the ADAS systems containing all the block functionality of the AV such as AEB, ACC, advanced park assist (APA), direction steering automatic emergency (AES). Since the simulator can replace AD architecture components, it can improve ADAS functionality by providing testing on simple representations and better planning decisions. Figure 4.10 shows how ADAS applies fusion on the raw data and provides the higher-level blocks of data that separate detected objects based on their type.

Although the number of AV simulation tools has increased in recent years, it is very challenging to select the best tool for a specific development. Many factors are desirable in simulators such as open-source, cross-platform, customization, and documentation. In the table 4.1, we present a summary of the specific simulators and their features and functionalities used for AD.

Several studies exist for the evaluation of AV simulators [118] and for a sys-

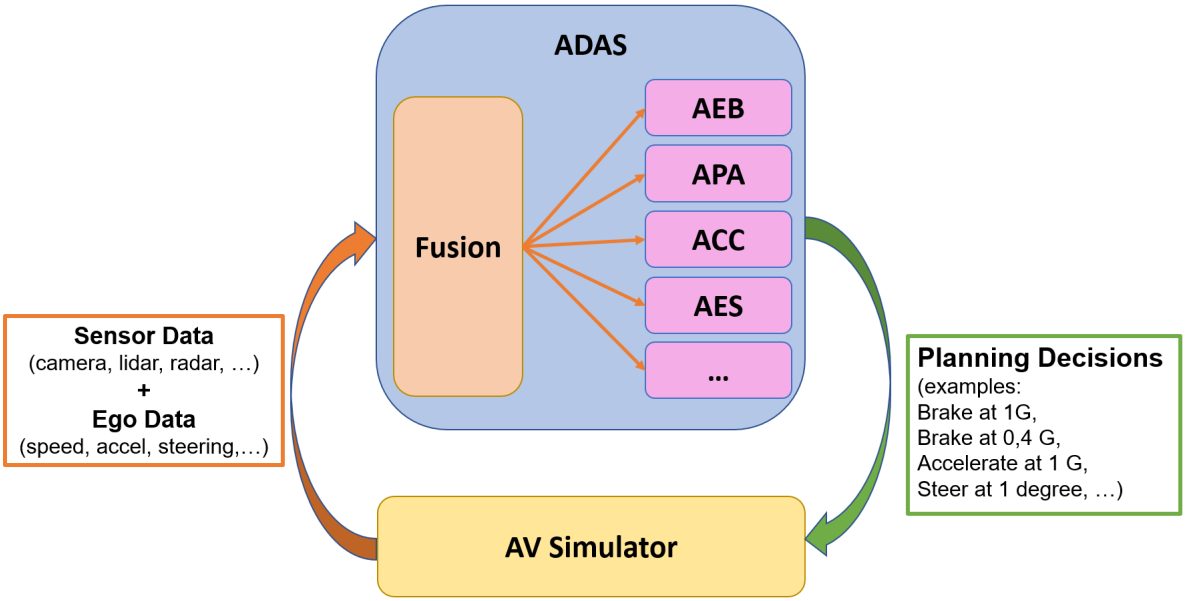

Figure 4.10: Simulation in the loop for ADAS.

tematic review of the perception system and simulators [119]. Simulators have become standard tools for many companies when considering a new installation or a change in production. Although these driving simulators are not the objective of the thesis, the proposed approach makes it possible to generate code compatible with the scripting language of the chosen simulator. It should be noted that the user can apply the EPSAAV tool to any simulator.
In Renault Group, SCANeR [117] is one of the simulators used to simulate scenarios. Developed for automotive experts, SCANeR Studio is designed to meet the specific needs of dynamic simulation professionals. They also use another tool called *FusionRunner* which runs a resimulation of real or generated scenarios from SCANeR. In the following Section, we talk about the advantages of *FusionRunner* for the thesis.

4.3.3 *FusionRunner* debugger

The main objective of *FusionRunner* is the development of algorithms and on-target embedded code, the development of multi-sensor fusion analysis, and debugging tools in the context of vehicle driver assistance. It is an optimized tool to better analyze and debug fusion algorithms, algorithm development for ego tracking using sensors, algorithm development based on artificial intelligence, and continuous resimulation with real perception. It traces input signals from real recordings or simulated ones from SCANeR simulator, and executes data fusion algorithm by replaying open-loop software as shown in Figure 4.11. An example of a SCANeR simulation is when we are testing a scenario with an input

Table 4.1: Summary of specific simulators and their features used for AD

Simulator	Scripting Language	License
Carla[80]	Python	GPL/Open Source
SIMLidar[111]	C++	GPL/Open Source
DeepDrive[112]	C++, Python	GPL/Open Source
Webots[82]	C/C++, Java, Python or MATLAB	GPL/Open Source
Udacity[113]	C++, Python	GPL/Open Source
AirSim[114]	C++, Python, C#, Java	GPL/Open Source
Carcraft/Waymo[115]	C++	Restricted
Helios[116]	Java	GPL/Open Source
SCANeR[117]	C/C++, C#, LabView, Python, Matlab/Simulink, FMI, RTMaps	Restricted

of braking at 1G or accelerating at 1G, etc., we want to know what the output will be. Both scenarios provide sensor data (camera, lidar, radar, etc.) and ego data (speed, acceleration, direction, etc.). This data is the input to the fusion. *FusionRunner* works for ACC and AEB feature components. This can provide

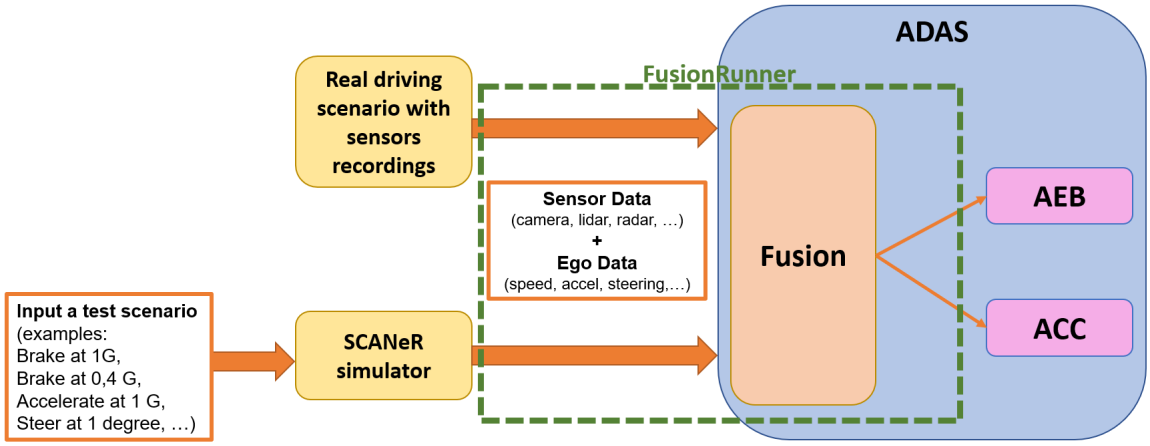

Figure 4.11: *FusionRunner* integration in the ADAS.

greater autonomy and complexity, emphasizing their perception and mapping subsystems [120]. *FusionRunner* is a general tool in which we can implement data fusion of different types of sensors. It executes tracking and applies machine learning standards for classifications. It can take into account several types of input formats, executes and displays fusion outputs using internal signals, and checks if everything is consistent.

4.3.3.1 Why using *FusionRunner*?

FusionRunner has many advantages in AD resimulation.

1. First, it runs the perception algorithm of Renault and performs sensor data fusion. It is an accessible tool for the thesis.

2. It can provide a re-simulation of real or generated scenarios (e.g., from SCANeR simulators).

3. It provides a wealth of environmental information. It can offer raw data and fused data, and already incorporate useful computed quantities such as TTC, Path Prediction, critical target selection.

4. It can take into account several sensors.

5. It can be used offline to reevaluate, and online.

6. It helps to visualize the inputs, internal data, and outputs of the Renault Fusion data which allows evaluating the performances of the algorithms.

7. The processing time in resimulation is shorter than in real-time on a standard central processing unit (CPU). In Renault, using five CPUs with eight cores, we can replay 10000 kilometers of driving data within half an hour. Additionally, it is sometimes necessary to reprocess the scenario to access the specific of an entire data. In this case, the system must avoid reprocessing due to the amount of time it takes and the cost it generates. *FusionRunner* provides reprocessing in a visual interface that optimizes the time and uses curves and graphs.

4.3.3.2 *FusionRunner*: a visualization tool for debugging

It features a command-line interface in C language as well as a scripting interface for the Graphical User Interface (GUI). We use the command-line interface to record data, or even playback recorded data in *FusionRunner*. The GUI consists of several dialog windows. The first window in the Figure 4.12 is the Control window designated as an interface. The control window provides:

- A **File** menu to open and close scenarios.
- An **Options** menu to access settings and open other windows.
- A **Perfs** menu to assess signals performance.
- A **Signal** menu to display signal curves for each module.
- A **Help** menu that opens a help page.

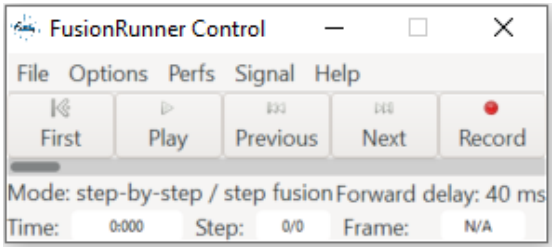

Figure 4.12: *FusionRunner* Control window.

- A set of buttons to control the scenario execution.
- A scrollbar that allows the user to navigate through the current scenario.
- Information about current playback mode, single-step mode, and forward delay.
- Text entry boxes for time, current step (fusion iteration), and current frame (context) that support user entries.

The time indicated in the *FusionRunner* control window always refers to the fusion time. However, the fusion call is only performed if the current time is strictly greater than the fusion call time where the fusion has not yet been called. When a scenario is loaded, *FusionRunner* first calculates the start and end times of the simulation. The simulator stores all the data so that we can come back to each step if ambiguity is placed.
The Fusion Context window in Figure 4.13 displays the current frame of the loaded scenario. We can compare the video to the measured data which enables safety assessment.

Figure 4.13: Fusion Context View window.

The Fusion Display window in Figure 4.14 illustrates the fusion outputs of the most recently executed fusion calls. This also displays the current open scenario,

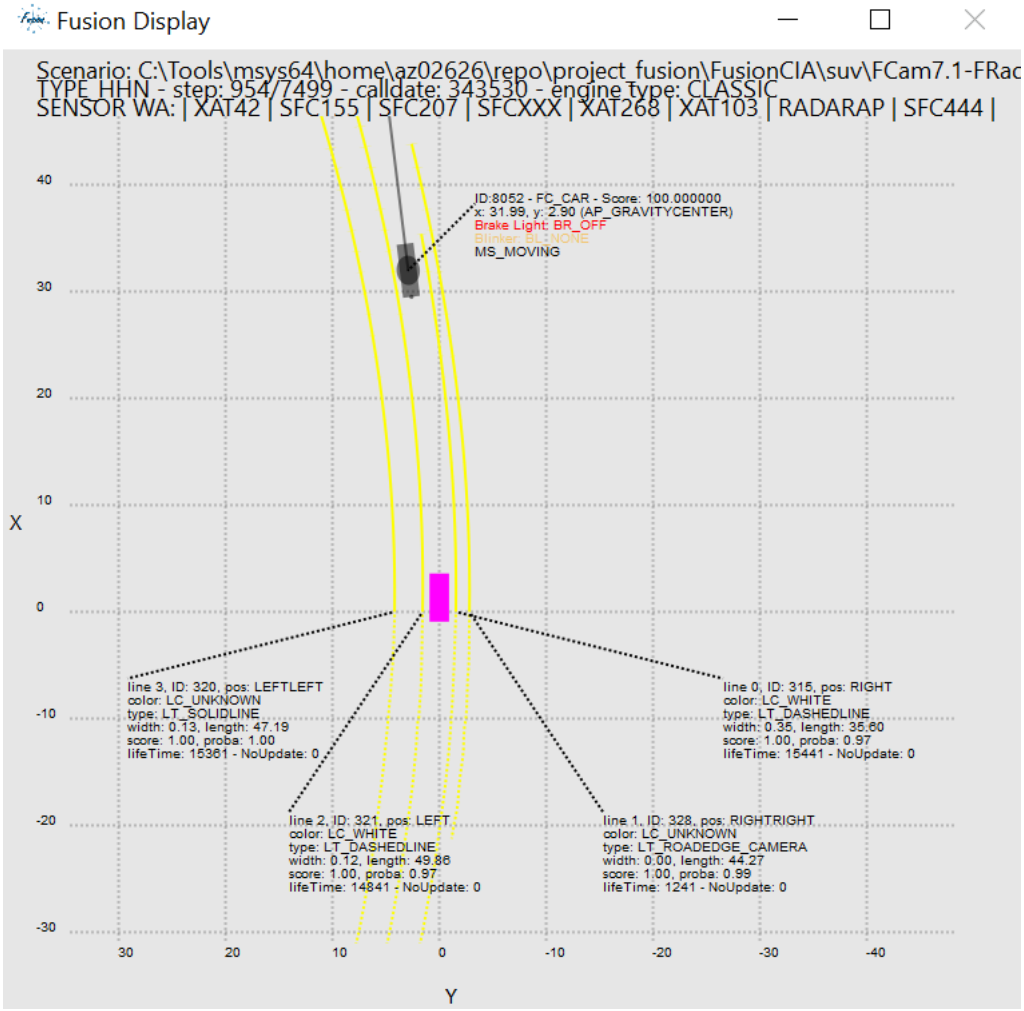

Figure 4.14: Fusion Display window.

current fusion step, and call date. Two actions can be performed on the Fusion Display window:

1. Zoom in / Zoom out: using the mouse scroll wheel.

2. move displayed items using mouse right-click.

These essential windows provided in *FusionRunner* help the safety engineer stop program execution and carefully check for ambiguities that arise from their formal safety specifications. *FusionRunner* allows us to examine the values of variables, the execution of generated safety module, and to inspect violations of specific functions and rules. We dive deeper into the generated safety verifier module in the following Section.

4.4 C Code generation using EPSAAV language

The proposed DSML allows C-code generation, which we call *Safety Checker module*, compatible with the simulator used. As mentioned before, for the thesis, we use *FusionRunner* which gives access to information needed for safety instead of adding new algorithms. After describing safety rules in the Rule-Based Planner using **EPSAAV specifier**, EPSAAV tool generates a C code that shares the same language with the *FusionRunner*. The *Safety Checker module* is generated from the user-defined environment and rules as shown in Figure 4.15. This code

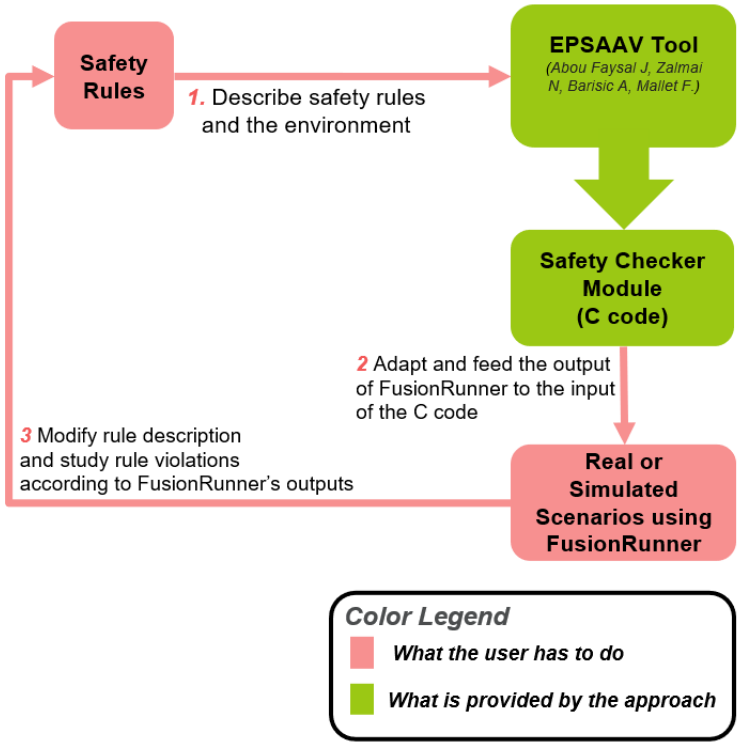

Figure 4.15: *Safety Checker module* generation process with EPSAAV language.

is then fed and interfaced with the output of the debugger. Yield allows rules to be changed and violations to be investigated according to *FusionRunner*'s output. This work follows an iterative process where a safety expert can refine and improve his requirements based on the analysis of the outputs from the resimulation.

In this Section, we show the *Safety Checker module* that is generated to assess and review safety. We also show how we visualize the safety signals' performance using a window we added to give us feedback on safety rules.

4.4.1 *Safety Checker module* integration

The *Safety Checker module* enhances the effectiveness and efficiency of the safety rules verification process, enabling the safety expert to meet the specific demands required for Functional Safety and SOTIF certifications. With the *Safety Checker module*, the safety engineer can automatically detect interference between software elements with different ASIL by checking formal rules and defined requirements. The *Safety Checker module* contains automatically generated and verified

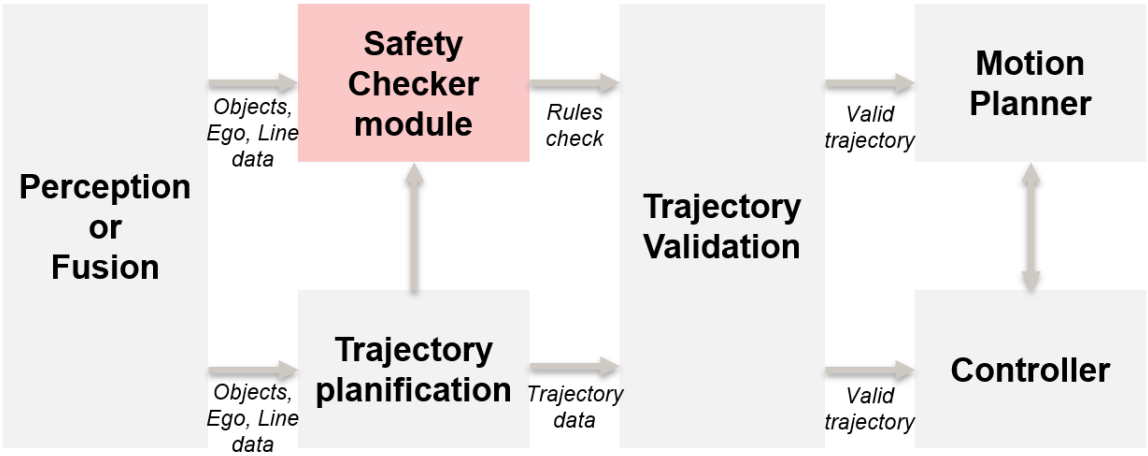

Figure 4.16: *Safety Checker module* implemented in AD software architecture.

C code that is error-free as the experts are actively working on specification designation, allowing them to maximize their time and productivity while developing the highest quality applications and safety. Table 4.2 shows the automatically generated C code with their headers. The extensions used are for defined files created by the user to identify the environment.

Table 4.2: C code generated from our framework based on EPSAAV language.

C code	Header	extension of the input file	Linked object
ActionLibraryName	ActionLibraryName	**.actions**	ActionLibrary
AlertLibraryName	AlertLibraryName	**.alerts**	AlertLibrary
PropertyLibraryName	PropertyLibraryName	**.prop**	PropertyTypeLibrary
SceneName	SceneName	**.scene**	Scene
Safety_Checks	Safety_Checks	**.rbp**	RuleBasedPlanner

The *Safety Checker module*'s automated features can save hours of the development process, eliminating the need to perform manual code checks by hand. All

the generated code is compiled and verified so that the user has only one task to interface with the fusion output described in Section 4.5.

- ActionLibraryName.c and AlertLibraryName.h are about enumerating actions and alerts in defined structures and instantiating process functions for both.

- ActionLibraryName.c and AlertLibraryName.c contain switch cases in process functions with all actions and alerts respectively to be printed to the user. For this thesis, actions have an informative task, but for future outlook, actions could be related to motion planner and controller, as shown in Figure 4.16.

- PropertyLibraryName.h renames properties as data types such as **propertyName_t**, and lists their states. It also declares a function for each property in this form: *propertyName_t propertyName_check();*

- PropertyLibraryName.c contains functions for each property and is detailed with instructive comments. All generated comments must be filled in by the engineer to bridge the gap between rules inputs and the debugger outputs. They contain if conditions for each state as seen in Figure 4.18.

- SceneName.h is composed of instantiated functions and structures. For structures, it defines each objectType as a data type and lists the parameters defined in the structure. For functions, it instantiates the parameter functions for each objectType in the following form: **paramType_t** *get_objectType_param();*. It also contains a function named *fusionDatatoScene()* which takes fusion data as a parameter.

- SceneName.c instantiates *fusionDatatoScene()* by calling functions for each objectType to get its parameters. This part is also to be filled in by the user to interface the parameters with the fusion data as shown in Figure 4.20.

- Safety_Checks.h contains **Goal_t** structure composed of three functions: (1) *isTriggered()* to include safety rules, (2) *execute()* to call actions, and (3) *raiseAlarm()* to trigger alerts.
  We also instantiate trigger conditions for each goal *trig_goalx_conditiony()*, raising alarms *alarm_goalx_conditiony()* and *execute_goalx_conditiony()* for the actions as different functions for each condition in a goal.

- Safety_Checks.c details *init_goals()* function that initiates all goals with conditions, and *trig_goals()* that triggers them by applying priority or parallel execution. If Goal1 has priority over Goal2, we test if Goal1 is false before triggering Goal2. It also details (*trig_goalx_conditiony()*) that transforms

the logical expressiveness of each condition into an if statement using logical operators. It also performs runtime and alarm functions based on what the safety engineer has set.

4.4.2 Visualization of *Safety Checker module* performance signals

The visual safety appraisal can be used to assess the safety engineer's decision to determine visual communications needs. The use of *FusionRunner* was initially designed around the process of using driving simulators and was structured around the accessibility of the graphical interface.

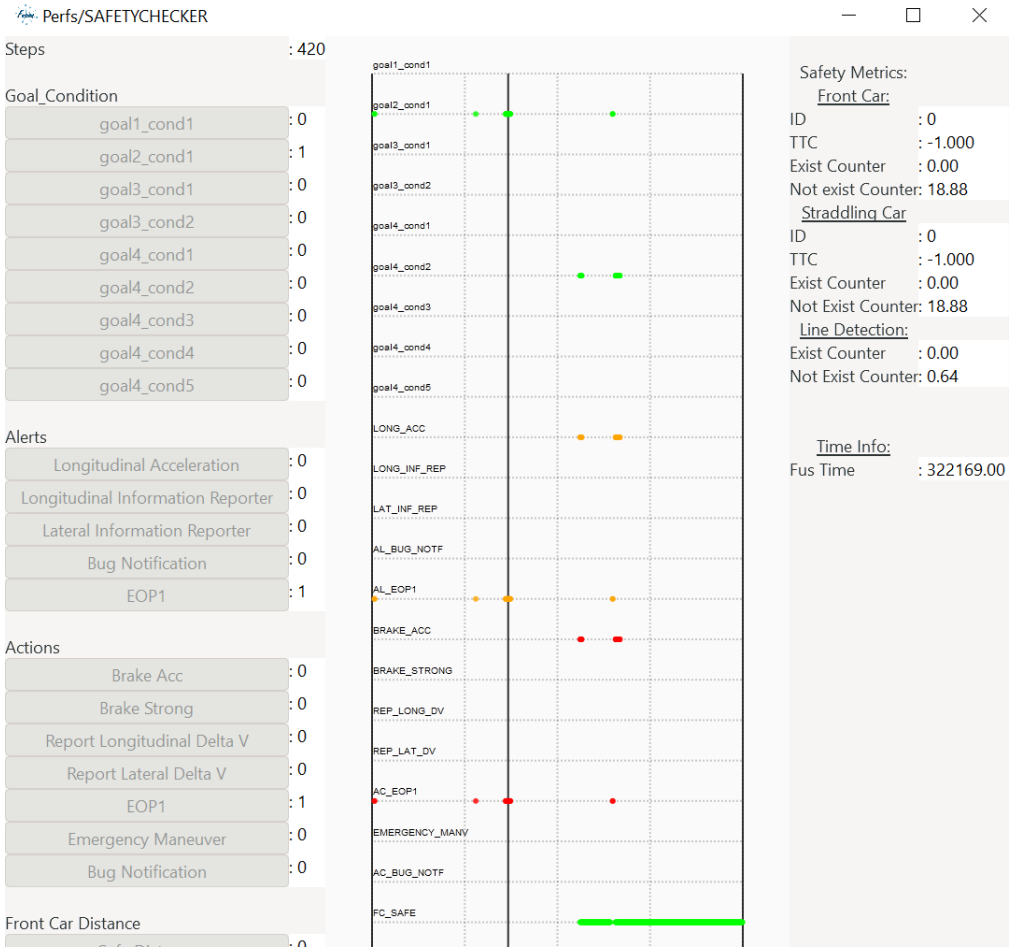

Figure 4.17: SAFETYCHECKER window to view safety measures, all goals with triggered actions, and alerts and properties.

To facilitate the safety evaluation procedure, we have created a window to visualize all the performances of the safety signals. The addition of a graphical interface simplifies debugging and represents safety metrics and shortcuts to

increase assessment productivity. We can manipulate back in time to check for ambiguities of violations triggered at a certain step. The SAFETYCHECKER window in Figure 4.17 defines the rules, properties, and parameters (safety metrics) declared by the engineer. It shows the interface created in the Perfs menu to visualize the use case.

4.5 Interfacing the *Safety Checker module* with the *FusionRunner* to assess safety

As mentioned above, the second step of the user process depicted by Figure 4.15 is to interface the *FusionRunner*'s output with the C code input in the *Safety Checker module*. The software engineer has two files to fill:

1. PropertyLibraryName.c where it needs to fill in the if statements with the correct thresholds by calling functions from SceneName.c as shown in Figure 4.18. An example of filling the thresholds and statements from Figure 4.6,

```
property1_t property1_check() {

    /*TODO User Adaptation : functions declared in SceneName.c with thresholds

    if () {
        return state1_property1;
    } else {
        return state2_property1;
    }
    */
    return state1_property1;
}
```

Figure 4.18: PropertyLibraryName.c containing check functions with If statements.

   is taking into consideration the Time To Collision (TTC) parameter by calling the function **get_preceding_vehicle_ttc_min()**. The filling form is shown in Figure 4.19.

2. SceneName.c in which engineer should provide fusion data as parameter and get necessary parameters for each objectType as shown in Figure 4.20.

These functions need to be manually filled by the software engineer to choose the right getter functions to defined thresholds. In case the safety engineer does not specify any threshold, the software engineer will apply his thresholds based on the states declared. When the interface is completed, the *Safety Checker module* becomes part of the ADAS to improve and assess the safety of AEB and ACC functions. Once all the rules are well specified, *Safety Checker module* can override these features to guarantee when to brake or accelerate.

```
front_car_distance_t front_car_distance_check() {

    static const float safe_distance_threshold = 2.0f; //safe_distance>=2.0s
    static const float acc_distance_threshold = 1.2f; //acc_distance>=1.2s
    static const float strong_braking_distance_threshold = 0.8f; //strong_braking_distance>=0.8s

    if(get_preceding_vehicle_ttc_min()==FLT_MAX){
        return not_exist_front_car_distance;
    }else if(get_preceding_vehicle_ttc_min()>=safe_distance_threshold){
        return safe_distance_front_car_distance;
    }else if(get_preceding_vehicle_ttc_min()>=acc_distance_threshold){
        return acc_distance_front_car_distance;
    }else if(get_preceding_vehicle_ttc_min()>=strong_braking_distance_threshold){
        return strong_braking_distance_front_car_distance;
    }else {
        return imminent_collision_distance_front_car_distance;
    }
}
```

Figure 4.19: Example of interfacing the If statements with a function in PropertyLibraryName.c.

```
static Ego_t Ego;
static ObjectType_t object_type_x;

static void Ego_getParameters(/*TODO User Adaptation : Parameter FusionControlData */);
static void object_type_x_getParameters(/*TODO User Adaptation : Parameter FusionControlData */);

void fusionDatatoScene(/*TODO User Adaptation : Parameter FusionControlData */) {
    Ego_getParameters(/*TODO User Adaptation : Parameter FusionControlData */);
    object_type_x_getParameters(/*TODO User Adaptation : Parameter FusionControlData */);
}

static void Ego_getParameters(/*TODO User Adaptation : Parameter FusionControlData */) {
    /*TODO User Adaptation : Link Fusion Output with Autogenerated Parameters in Scene.h */
}

static void object_type_x_getParameters(/*TODO User Adaptation : Parameter FusionControlData */) {
    /*TODO User Adaptation : Link Fusion Output with Autogenerated Parameters in Scene.h */
}
```

Figure 4.20: Comments to interface fusion data in SceneName.c.

4.6 Conclusion

In this Chapter, we have mentioned the perceptual challenges of sensors that can be enhanced by using parameters and properties introduced by the EPSAAV tool. We have validated the use of a simulator for the proposed approach. We coupled the simulator objective to the representative AD software architecture. Any simulation can replace perception and planning components that reduce processing time and provide access to pre-processed scenarios in a short time.
The proposed approach can be interfaced with any existing simulator. In this Chapter, we announced the use of *FusionRunner* which performs resimulation and debugging of generated or real scenarios. *FusionRunner* presents many advantages discussed earlier. It generates metrics that can be consulted via an interface to analyze at a glance the quality and performance of the software and to quickly identify potential problems.

FusionRunner calculates all necessary information such as braking and improves algorithmic performances. It gives us access to raw or fusion data. However, it remains a tool to maintain because it is coded in C/C++, which requires a lot of effort. This is where the proposed approach comes to the rescue by automatically generating code and replacing feature components. It enriches the simulation by creating a safety branch to validate the development process and test ambiguities in the scenarios.

Chapter 5

From Safety Rules to Satisfaction Checking

5.1 Introduction

To enhance the development of safety systems, MBSE has been used to provide early validation and verification for DSML. Formalizing specifications helps catch mistakes and study the properties of the solutions. It also clarifies concepts and paper proofs by complementing them with automatic strategies. ISO 26262 extends the concept of Safety Integrity Levels (SIL) proposed in IEC-61508 and is adopted and redefined as Automotive Safety Integrity Level (ASIL). It examines the functional safety requirements for all the different electrical and electronic systems of a vehicle. ASIL-D is an automotive hazard classification that is part of the functional safety and represents the highest level of risk management, so components or systems developed for this level are manufactured to the strictest safety requirements that highly recommend formalization. We focus on safety requirements that must be described by an appropriate combination of formal methods that specify the concept phase for automotive applications. The verification of these systems is done by providing formal proof on an abstract mathematical model. Several DSML approaches apply software verification techniques and validate the implementation by applying automatic or semi-automatic solvers (Section 5.2). As global verification often leads to undecidable problems, most problems are usually reduced to domains where we have solvers available. Because of the expressiveness of EPSAAV, that is driven from the safety domain, we decided to reduce our safety rules to a Boolean Satisfiability Problem (SAT). This is a first step to convince the industry of the benefit of moving from a paper document to a computer model as it immediately opens a wide range of validation and verification possibilities. In this thesis we use a SAT Solver (SAT4J relying on the Java language) to detect inconsistencies between safety rules that are difficult to detect when the specification is too large and certainly very error-

prone if the process remains paper-based. Besides, this encoding of our rules as Boolean Predicates gives a formal interpretation (by transformation) to the EPSAAV language.

In this Chapter, we provide the requirements for using the SAT solver in Section 5.3. We mention the need of the solver by giving rules inconsistencies examples in Section 5.4, and what are the three implementation steps in the tool to automatically generate Java code in Section 5.5. This compatible code is then transmitted to the solver and depends on the formal rules described by the safety expert. This makes it possible to detect inconsistencies and check the validity and completeness of the rules. These inconsistencies can result from a requirement or specifications executed in parallel or sequential. To assess that, we present tests to verify SAT solutions. This is all described in the Section 5.6. If the safety engineer wishes to modify the environment, the code automatically adapts to his modifications.

5.2 Satisfiability solvers

In the early 2000s, Satisfiability Modulo Theories (SMT) has built on the success of SAT solvers and on their efficiency to extend the expressiveness of what can be verified by defining ad-hoc theories [121, 122]. While there is a real opportunity in EPSAAV to benefit from SMT solvers we decided to limit our study to SAT solvers at it is already a big step compared to the state of practice and it is sufficient to prove our points. However, it is very clear that relying on SMT would open a vast domain of further verifications that could be treated as future work but that remains out of the scope of this thesis.

Nowadays, most verification tools depend on solvers which are now very popular tools for solving different types of problems. We can cite safety verification studies on the properties of finite state machines using a SAT solver [123]. Recent breakthroughs in their development have also resulted in a great advance in the relevance of SMT solvers, leading to the development of many different industrial applications in the fields of software verification, model-based testing and debugging [83], verification of models, and use case generation [124, 125]. SMT solvers that solve more advanced theories are necessarily incomplete or even slower than SAT solvers [126]. Most SMT solvers use a state-of-the-art SAT solver to evaluate whether an SMT instance is satisfiable or not. This is why, within the scope of this thesis, we pick out for the beginning the suitable SAT solver and integrate it into the framework. The SAT solver gives us a certainty of validity of our results. For future perspectives, SMT could be integrated to provide more complex cases. The formal rules defined are based on simple logical operations which can be defined using the SAT Solver.
We describe in detail the SAT solver and the steps for its integration.

5.3 SAT solver

Satisfiability is the problem of determining whether the variables of a given Boolean formula can be assigned in such a way that the formula evaluates to true. If no such assignment exists, it means that the function expressed by the formula is false for all possible variable assignments. In the latter case, we say that the formula is satisfiable; otherwise, it is unsatisfiable. The SAT solver addresses this problem and is known for computational complexity, representing the first decision problem to be proven NP-complete [84, 127]. It has been used in many practical applications and is remarkably motivating MBSE [128]. It serves as a trusted kernel checker for verifying results of other untrusted verifiers such as BDDs, model checkers, and SMT solvers [129]. Almost all SAT solvers include time-outs, so they terminate in a reasonable time even if they cannot find a solution. Different SAT solvers find different instances easy or hard, and some excel at proving unsatisfiability, and others at finding solutions [130]. Their tools excel at finding inconsistencies by interpreting domain specifications as a logic problem [131]. SAT4J is the java library used for solving Boolean satisfaction and optimization problems. Being in Java, the promise is not to be the fastest one to solve those problems, but to be fully featured, robust, user friendly, and to follow Java design guidelines and code conventions solving SAT problems. The library is designed for flexibility, by using heavily the decorator and strategy design patterns. Furthermore, SAT4J is open source, under the dual business-friendly Eclipse Public License and academic friendly GNU LGPL license [132].

5.4 Rules contradiction

To help the safety engineer catches inconsistencies in his rules, we included a SAT solver in our process. Let us see the following example:

$$\begin{aligned} RBP = (&Goal1Condition1 \implies action1) \wedge \\ &(Goal1Condition2 \implies action1) \wedge \\ &(Goal1Condition3 \implies action1) \end{aligned} \tag{5.1}$$

The first interpretation satisfies the formula RBP if (Goal1Condition1 $\implies$ action1), (Goal1Condition2 $\implies$ action1) and (Goal1Condition3 $\implies$ action1) are true. A second interpretation does not satisfy the formula if one of them is false. We can take the real example described previously in Figure 3.14 for the safety assessment with the following conditions:

$$\begin{aligned} Goal1Condition1 &= \text{follow the PV;} \\ Goal1Condition2 &= \text{respect speed threshold;} \\ Goal1Condition3 &= \text{respect safety distance of 2s;} \\ \text{and } action1 &= \text{light acceleration.} \end{aligned} \tag{5.2}$$

At first sight, there is no contradiction in this example among the rules because we have different conditions for these goals. However, let us suppose that we have a policy system having the following properties:

$$\begin{aligned} RBP2 =& (Goal1Condition1 \implies action1) \wedge (Goal1Condition2 \implies action1) \\ & \wedge (Goal1Condition3 \implies action1) \wedge (Goal2Condition1 \implies action2) \end{aligned} \tag{5.3}$$

$$\begin{aligned} Goal2Condition1 &= \text{respect a TTC for every VRU;} \\ \text{and } action2 &= \text{emergency braking.} \end{aligned} \tag{5.4}$$

Conflict occurs with RBP2 because it satisfies action1 and action2. Satisfying them means that the AV is allowed to accelerate and decelerate at the same time. Imagine the case study when the RBP contains numerous goals that are composed of multiple conditions and the safety expert needs to be sure that there is no inconsistency. An obvious way to solve the SAT problem is to traverse the truth table for the expression as seen in tables 5.1 and 5.2.

Table 5.1: Truth table for the implication expression of Goal1Condition1 in RBP2.

Goal1Condition1	action1	Goal1Condition1 ⇒action1
0	0	1
0	1	1
1	0	0
1	1	1

The logical contradiction in the previous example is clear and easy, but as the system becomes more complex, we need truth tables to enumerate all combinations of inputs and define if a solution exists. We propose an implementation process to deploy the SAT solver which tests whether the change still has a conflict arisen and determines which subjects and resources are involved in the inconsistency. Our proposed language gives the possibility of modifying these conflicts to eliminate all possible inconsistencies.

5.5 Phases of implementation tasks to deploy the SAT solver

Executing formal requirements using operations is an important technique for requirements validation and rapid prototyping. We present an efficient and fully automatic approach to run the code generated from the rules and specifications which then uses a SAT solver as shown in Figure 5.1. The requirement is trans-

Table 5.2: Truth table for the RBP2 expression.

Goal1Cond1 ⇒action1	Goal1Cond2 ⇒action1	Goal1Cond3 ⇒action1	Goal2Cond1 ⇒action2	RBP2
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

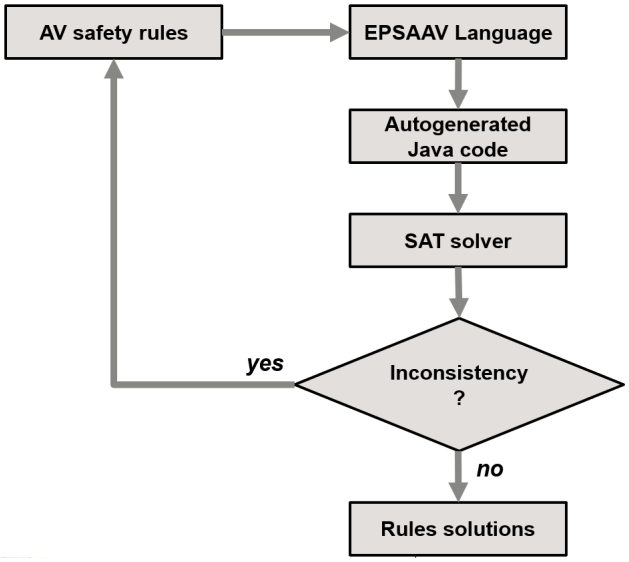

Figure 5.1: Phases of implementation tasks to study inconsistency using the DSML proposed and the SAT solver.

lated into a logical formula and later into a satisfiability problem. Based on the state of the system in which the operation is called and the arguments of the operation, an out-of-the-box SAT solver calculates a new state that satisfies the

conditions of the operation. An effort is made to keep system state changes as small as possible. We present a tool to generate Java method bodies for operations specified in requirements. A SAT solver decides whether a Boolean formula is satisfiable. If there is an inconsistency present, the engineer can see its origin and then modify the rules. In other words, three commitments are developed and given by the process:

1. Translating rules to Boolean formulas
2. Testing inconsistencies
3. Verifying solutions

5.5.1 Translating rules to Boolean formulas

First, domain experts have to describe the safety rules executed in parallel or sequentially based on the environment (scene, parameters, and properties) and the behaviors (alarms and alerts), using formal syntax. By translating rules to Boolean formulas using EPSAAV language, SAT solvers can be used to determine valid system states for the animation. Boolean formulas concern the following logic operators in the table 5.7. A formula contains binary variables that are connected by logical relations. Boolean operators are used as standard functions that take one or more binary variables as input and return a single binary output. Each Boolean operator is defined by a truth table in which we enumerate every combination of inputs and define the output. Common logical operators include:

1. The AND operator that is written as $\wedge$ and takes two inputs and returns true if both are true as seen in table 5.3.

Table 5.3: Truth table for the AND logic operator.

Goal1	Goal2	$Goal1 \wedge Goal2$
0	0	0
0	1	0
1	0	0
1	1	1

2. The OR operator that is written as $\vee$, takes two inputs and returns true if at least one of them is true as seen in table 5.4.
3. The IMPLICATION operator that is written as $\Rightarrow$ and evaluates whether the two inputs are consistent with the statement if..then. Its truth table is shown in Figure 5.1.

Table 5.4: Truth table for the OR logic operator.

Goal1	Goal2	$Goal1 \vee Goal2$
0	0	0
0	1	1
1	0	1
1	1	1

Table 5.5: Truth table for the NOT logic operator.

Goal1	$\neg Goal1$
0	1
1	0

4. The NOT operator is written as $\neg$ and takes one input and returns true if the input is false and vice-versa as seen in table 5.5.

5. The EQUIVALENCE operator $\Leftrightarrow$ operator and takes two inputs and returns true if the two inputs are identical and returns false otherwise as seen in table 5.6. It can be decomposed as two implications as seen in table 5.7.

Table 5.6: Truth table for the EQUIVALENCE logic operator.

Goal1	Goal2	$Goal1 \Leftrightarrow Goal2$
0	0	1
0	1	0
1	0	0
1	1	1

We aim to establish whether there is any way to set these variables so that the formula evaluates to true. We rely on those multiple basic formulas adaptable to the SAT solver, presented and implemented from [133]. We chose to integrate its code for the ease of properties name mapping. As seen in the table 5.7, to pass from the Example using a logic operator to its corresponding Boolean formula, the EPSAAV generator eases this affiliation.

All the rules presented in the rule-based planner must be translated into these basic formulas. A boolean logic formula takes a set of variables that can be true or false and combines them using Boolean operators returning true or false. The RBP and RBP2 formulas defined previously in equations 5.1 and 5.3 are examples of rules using logic operators with 4 and 6 defined variables respectively. For the second combination, we could evaluate this formula and see if it returns true or false. Notice that even for these simple examples it is hard to see what the answer is by inspection. This is why we work on generating Java

Table 5.7: Basic logical symbols integrated in the SAT solver.

Symbol	Read as	Example	Boolean formula
$\wedge$	and	$c = a \wedge b$	$IBoolSpecification.and(\text{“}c\text{”}, \text{“}a\text{”}, \text{“}b\text{”})$
$\vee$	or	$c = a \vee b$	$IBoolSpecification.or(\text{“}c\text{”}, \text{“}a\text{”}, \text{“}b\text{”})$
$\Rightarrow$	implies	$c = a \Rightarrow b$	$IBoolSpecification.implies(\text{“}c\text{”}, \text{“}a\text{”}, \text{“}b\text{”})$
$\neg$	not	$c = \neg a$	$IBoolSpecification.not(\text{“}a\text{”})$
$\Leftrightarrow$	equivalence	$c = a$	$IBoolSpecification.implies(\text{“}c\text{”}, \text{“}a\text{”})$
			$IBoolSpecification.implies(\text{“}a\text{”}, \text{“}c\text{”})$

code compatible with the SAT solver using these formulas. The engineer does not spend time in hand-coding making fewer errors, which enables him to save time on implementing larger and more rules. In previous Chapters, we detail

```
WHEN{
    OR(
        propertyType "EgoProperties.front_car_distance" is
        "EgoProperties.front_car_distance.acc_distance",
        propertyType "EgoProperties.straddling_car_distance" is
        "EgoProperties.straddling_car_distance.acc_distance"
    )
    action "libActions.brake_acc"
    alert "LibAlerts.longitudinal_acceleration"
}
```

Figure 5.2: Example of a safety requirement defined by the expert using the EPSAAV tool.

the DSML development part and describe the tool based on a meta-model that facilitates the environment's description and generates monitors such as code. Figure 5.2 is an example of a goal definition where the system shall trigger a *brake_acc* alerting the engineer with the *longitudinal_acceleration* value. The policy is related to an OR logical symbol of an ACC *front_car_distance* and ACC *straddling_car_distance*, which implies triggering a brake and an alert to the engineer. Figure 5.3 represents the generation of the rules to Java code following the formulas defined in the table 5.7. This generation is done by taking into account the defined rules and transforming them into these Boolean formulas using a flexible and expressive dialect of Java code developed in Xtend projects. It is our intention to make use of the intense research on satisfiability solving for executing specifications. We have implemented the SAT4J library in Eclipse as seen in Figure 5.4, and tested the Java code generated.

The first phase consists of generating Java method bodies from goals that are compatible with the SAT4J library to solve SAT problems as shown in Figure 5.3. All logic operations for rules specification are translated to specific forms of coding. The "**is**" verb is translated from the tool by joining the property to the state using this form **state_property**. An example of ***front_car_distance***

```
static public void build_goal1_cond1(IBooleanSpecification spec) {
    spec.or("goal1_cond1", "acc_distance_front_car_distance",
            "acc_distance_straddling_car_distance");
}
```

Figure 5.3: Java code generation from a goal defined by the safety expert in Figure 5.2.

```
import fr.unice.lightccsl.sat.SAT4JSolverBuilder;

public class Test {
public static void main(String[] args) {
      SAT4JSolverBuilder builder = new SAT4JSolverBuilder();
      IBooleanSpecification boolSpec = builder.specification();
      BooleanSatisfactionProblem bsp = (BooleanSatisfactionProblem)boolSpec;
      MyNameMapper mapper = new MyNameMapper();
      bsp.setNameMapper(mapper);
```

Figure 5.4: Implementation of SAT4J library in Eclipse.

is ***acc_distance*** is translated to ***acc_distance_front_car_distance*** as seen in Figure 5.3. We follow this format to avoid duplicating same state if it exists twice such as **acc_distance** for both properties **front_car_distance** and **straddling_car_distance**.

5.5.2 Testing inconsistencies

The second phase consists of testing the inconsistency of the rules applying specific formulas by automatically generating specific controls for each rule. When the safety engineers decide to write their code, they can make a mistake in the order of the rules or even repeat their definition twice. To help them determine whether his RBP is consistent or not, several tests should be run. Before digging into these formulas, let us first understand what needs to be addressed. First test is to validate that solutions exist when the rule (or goal) is true as follows:

$$test(Goal) \begin{cases} true, & \text{rule valid} \\ false, & \text{rule invalid} \end{cases}$$

We can give a simple example using an AND operator:

$$\begin{aligned} Goal = (front\_car\_distance = safe\_distance) \wedge \\ (front\_car\_distance = not\_exist) \end{aligned} \tag{5.5}$$

This example shows that a front_car_distance should only have one state, either a safe_distance or not_exist. So; no solution for this rule, and the rule is then invalid. This test is important to see if the rule is valid or not, meaning that the associated triggering condition of a goal is satisfiable or not. After verifying that

the rule is valid, we can analyze all the solutions. It is then important to be able to see if there is a case in the truth table where two contradictory actions can be triggered at the same time. Another test is applied on two conditions within the same goal, or even on two goals if they contain only one condition each, to determine the type of execution:

$$test(G1, G2) \begin{cases} Priority\_execution \\ Parallel\_execution \\ Identical\_rules \end{cases}$$

We take the example in RBP2 in equation 5.3.

The Goals (or even Conditions in Goals) that trigger different behaviors (examples *action1*=light acceleration, and *action2*=emergency braking) are not satisfiable at the same time, then a priority must be given. If two conditions are exclusive then the order does not matter and there is no need for assigning a priority. This test is necessary to see if two same states that are present in the requirement have always the same value. We also apply test on the whole system regarding the actions, the alerts, and the coherence between the goals and the states. The system solution presents all possible solutions to the defined rules,

$$test(system\_solution) \begin{cases} true, & \text{system coherent} \\ false, & \text{system not coherent} \end{cases}$$

After applying the tests for the system, if we get solutions, the system will give us consistent output. If there is no solution for the system, then there is no consistent solution. These tests are performed on successive matched rules and for the whole system. This test is useful in case several conditions withing several (concurrent) goals would be satisfiable at the same time but leading to different actions. Given the specification, our framework generates a campaign of tests, each individually testing one particular kind of inconsistency (akin to unit testing). If one test fails, it points at a possible inconsistency within the rules that must be addressed by the safety engineer. We take the example of RBP2 in the equation 5.3 but G1 and G2 are executed sequentially. The *system_solution* will give us all the possible solutions consistent with the defined behaviors and properties. Since they are now executed in sequence, the system will present consistent solutions. If we test the Goals in RBP2 in parallel with the consistency of behaviors and properties, then the system will give all the inconsistencies present as solutions.

5.5.3 Boolean Satisfiability problem

The Boolean Satisfiability problem asks whether there is at least one combination of Boolean input variables that makes the Boolean logic formula True. When this is the case, we say the formula is satisfiable. SAT solvers algorithms can

be classified into two types. First, complete algorithms guarantee to return SAT or UNSAT. Second, incomplete algorithms return SAT or return UNKNOWN without providing an answer. The best case is when they find a solution that satisfies the expression. If they do not, then we can not draw conclusions. The SAT solver that constitutes an algorithm for establishing satisfiability, takes the Boolean logic formula as input and returns three types of SAT solutions for the SAT problem as seen in Figure 5.5.

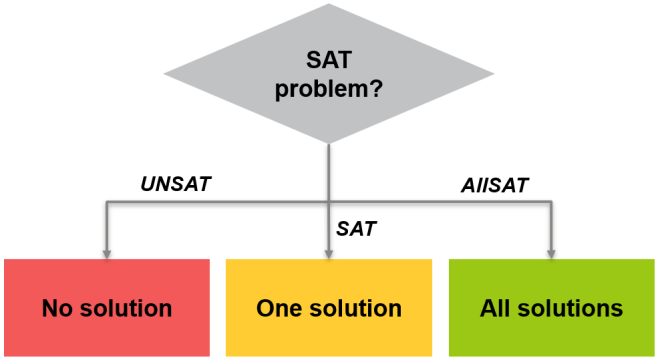

Figure 5.5: SAT problem and solutions.

If it is a SAT problem, we have at least one solution verification if it finds a combination of variables that can satisfy it. For instance, RBP2 in table 5.2 has a solution if all the variables are true. Otherwise, the conditions defined in the rule have no solutions, which means UNSAT can demonstrate that no such combination exists. In addition, it may sometimes return without an answer if it cannot determine whether the problem is SAT or UNSAT. ALL SAT solutions mean computing all the configuratio of input variables that make the formula True. Depending on solvers, it may be very expensive to compute all SAT.

5.6 Java code generation for inconsistencies study

The inconsistency is the lack of consistency such as statements can be not compatible with another one. They contain incoherent or illogical elements in thought or actions. As previously introduced, we have three phases for testing inconsistencies on the rules: the validty of each rule, the consistency between the rules, and the consistency for the whole system to find all possible coherent solutions. Requirements have a very important role to play in a safe and reliable system. They provide general information about AV functionality, for example, they may indicate that the car is facing a perception error or that it should apply a full brake or even accelerate for comfort aspects. The requirements must always be followed because they are at a higher level than ordinary policies. So if there is a conflict between the requirements and the policies, the policies should change,

not the requirements.
Using the EPSAAV tool, the safety engineer can manipulate these policies to meet a specific requirement and try to claim for inconsistencies by analyzing the solutions. With the increasing complexity of AV systems, it becomes very difficult to cover all unsafe cases. Therefore, a safety system may contain a loophole. As a solution, we integrate a formal way to describe safety requirements into the tool to raise the existence of soundness rules. Soundness is the property of only being able to prove true things. Completeness is the property of being able to prove all true things. So a given logical system is sound if and only if the inference rules of the system admit only valid formulas. The SAT Solver can also be applied to specific tests to meet any requirements. This is why we differentiate three types of tests that are generated for the engineer to test his requirements.

- testing validity of internal rules,
- testing consistency of sequential and parallel rules with each other,
- testing of the whole system considering properties and goal coherence.

These tests are generated in three different Java files respectively to the previous testing phases:

- Sat4jRules.java is a Java file that contains the rules translated to Boolean forms,
- Sat4jRulesConsistency.java relies on the previous file to test whether some conditions are SAT or UNSAT, in turn. It also combines conditions from different goals to see which ones are compatible or not leading to a different recommendation depending on whether those conditions are sequential, or parallel, with or without priorities.
- and Sat4jSystemConsistency.java that helps verify solutions of the system and the goals coherence with the properties coherence. It combines all rules and actions of all the goals to see whether globally the system is satisfiable.

5.6.1 Testing validity of each rule in the rule-based planner

The EPSAAV tool helps the analysis of safety requirements in an automatic process. Using the Boolean satisfiability problem, we can find a solution that satisfies the expression. We can also check all the solutions that exist to analyze the choice of priorities. As we said earlier, a specific Boolean formula should be generated automatically from the rule-based planner.
As defined before, a Goal can be composed of multiple Condition. A Condition starts with a *WHEN*, and ends with an *Action* and an *Alert*. Consider

an example of rules defined in a rule-based planner in Figure 5.6. Goal1 regroups two conditions. The first one requires a deceleration when there is an emergency distance with the pedestrian detected (pedestrian_tracking is detection). The second one, which is the second *WHEN*, is a possibility of accelerating when the front car does not exist and has no emergency distance with the EV (neither strong_braking_distance nor imminent_collision_distance). Note that *OR(a,b,c)* can be also read as *(a OR b OR c)*; and *NOT(OR(a,b,c))=NOT(a OR bb OR c)=(NOT a) AND (NOT b) AND (NOT c)*. This example is inspired from the Figure 3.14 where two actions are contradictory and need to be specified as sequential rules. This case refers to the **EgoProperties** li-

```
GOAL goal1
{
GoalType Priority
WHEN{
        AND ( propertyType "EgoProperties.pedestrian_distance" is
        "EgoProperties.pedestrian_distance.emergency_distance",
        propertyType "EgoProperties.pedestrian_tracking" is
        "EgoProperties.pedestrian_tracking.detection"
        )
    action "libActions.perform_deceleration"
    alert "LibAlerts.emergency_braking"
}

WHEN{
    NOT(
        OR(
            propertyType "EgoProperties.front_car_distance" is
            "EgoProperties.front_car_distance.strong_braking_distance",
            propertyType "EgoProperties.front_car_distance" is
            "EgoProperties.front_car_distance.imminent_collision_distance",
            {propertyType "EgoProperties.front_car_distance" is
            "EgoProperties.front_car_distance.exist" ifonlyif(
            propertyType "EgoProperties.front_car_tracking" is
             "EgoProperties.front_car_tracking.stable_tracking")
            }
        )
    )
    action "libActions.perform_acceleration"
    alert "LibAlerts.light_acceleration"
}
```

Figure 5.6: Definition of a goal with two sequence conditions containing specific logical operators.

brary which contains four properties: **front_car_distance**, **front_car_tracking**, **pedestrian_distance** and **pedestrian_tracking** as shown in Figure 5.7. Each of these properties is assigned to variables. The goal translated into Boolean formulas is shown in Figure 5.9. The first condition in **goal1** is translated to *build_goal1_cond1()* function, and consists of decelerating. The second condition in **goal1** is translated to *build_goal1_cond2()* function, and consists of accelerating. For the internal validity rule test, we do not consider the actions and alerts, but rather test the rules if solutions are correct or not. Both functions take the logic requirements defined by the safety expert and translate the logical operators into Boolean formulas defined in the table 5.7. For the first condition translated into *build_goal1_cond1()*, we put the result of the AND operator of **emer-**

```
state "pedestrian_tracking" CanBe "detection"
"not_confirmed"
"not_exist"
state "pedestrian_distance" CanBe "emergency_distance"
"not_exist"
"safe_distance" operator">=" value 2.0 unit "s"
state "front_car_distance" CanBe "not_exist"
"exist"
"safe_distance" operator">=" value 2.0 unit "s"
"imminent_collision_distance"
state "front_car_tracking" CanBe "not_confirmed"
"not_exist"
"stable_tracking"
```

Figure 5.7: Definition of an Ego property library with four states containing variables.

gency_distance_pedestrian_distance and **detection_pedestrian_tracking** in **goal1_cond1**. For the second condition translated into *build_goal1_cond2()*:

- NEGATION of **goal1_cond2** which is the OR disjunction of **strong_braking_distance_front_car_distance**, **imminent_collision_distance_front_car_distance**, and the IFONLYIF operator seen in the next point and defined as **goal1_cond2_1**. The negation is commented later to not force the function to be false.

- **goal1_cond2_1** refers to the IFONLYIF. When **exist_front_car_distance** and **stable_tracking_front_car_tracking** are false, or when both are strictly true, **goal1_cond2_1** is true. This is why, the IFONLYIF uses the *build_Equiv* function (seen in Figure 5.8).
  In our case, a= **goal1_cond2_1** which is the name of the function, b=**exist_front_car_distance** and c=**stable_tracking_front_car_tracking** that can both be true or false at the same time. This function regroups all of these Boolean formulas:

```
static public void build_Equiv(IBooleanSpecification spec, String a, String b, String c) {
    spec.clause(a,b,c);
    spec.clause(b,a,c);
    spec.implies(a,c);
    spec.or(b+c,b,c);
    spec.forces(b+c);
}
```

Figure 5.8: Definition of the *build_Equiv* function replacing the IFONLYIF operator.

  - **spec.clause(a,b,c)** that is the equivalent to ¬a∧b∧c
  - **spec.clause(b,a,c)** that is the equivalent to ¬b∧a∧c
  - **spec.implies(a,c)** that is the equivalent to a⇒c or ¬a∨c

- **spec.or(b+c,b,c)** that is the equivalent to (b+c)=b∨c where the addition symbol is a concatenation method to rename a new variable that executes the OR operator between b and c
- **spec.forces(b+c)** that forces the concatenation b+c to be true.

```
static public void build_goal1_cond1(IBooleanSpecification spec) {
    spec.and("goal1_cond1", "emergency_distance_pedestrian_distance",
            "detection_pedestrian_tracking");
}

static public void build_goal1_cond2(IBooleanSpecification spec) {
    spec.not("goal1_cond2");
    spec.or("goal1_cond2", "strong_braking_distance_front_car_distance",
            "imminent_collision_distance_front_car_distance", "goal1_cond2_1");
    build_Equiv(spec,"goal1_cond2_1", "exist_front_car_distance",
                "stable_tracking_front_car_tracking");
}
```

Figure 5.9: Generation of Java function containing Boolean formulas compatible to the SAT4J library.

This is how all the rules are generated using the **EPSAAV generator** and transformed into functions interfaced with the SAT4J. Note that the translation of the states in each property becomes: **StateName_PropertyName**. In order to see solutions for this rendered objective, we need to force the goals to become true and test also their negation. To study the validity of the internal rules, we automatically generate for each condition two functions: *build_goalX_condY_True()* and *build_goalX_condY_False()*. The first one test the validity, the second one test the falsifiability. An example is shown in Figure 5.10.

```
static public void build_goal1_cond1_True(IBooleanSpecification spec) {
    Sat4jRules.build_goal1_cond1(spec);
    spec.forces("goal1_cond1");
}

static public void build_goal1_cond1_False(IBooleanSpecification spec) {
    Sat4jRules.build_goal1_cond1(spec);
    spec.not("goal1_cond1");
}
```

Figure 5.10: Generation of Java functions to test internal rule validity.

Before forcing the function to be true or false, we test the AND operator for **goal1_cond1**, we run the *build_goal1_cond1()* function and visualize three results in Figure 5.11. When we try to run functions in Figure 5.10 forcing true and false tests, we see that *build_goal1_cond1_True()* has one solution as seen in Figure 5.12, and *build_goal1_cond1_False()* contains two solutions as seen in Figure 5.13. So the function *build_goal1_cond1()* groups false and positive results. For the **goal2_cond2** (Figure 5.9), the *build_goal1_cond1_False()* function is the true one and vice versa.

```
[[-1, 2, -3], [-1, -2, 3], [1, 2, 3]]
Solution 1:
[-1, 2, -3]
1: goal1_cond1=0,
2: emergency_distance_pedestrian_distance=1,
3: detection_pedestrian_tracking=0,


Solution 2:
[-1, -2, 3]
1: goal1_cond1=0,
2: emergency_distance_pedestrian_distance=0,
3: detection_pedestrian_tracking=1,


Solution 3:
[1, 2, 3]
1: goal1_cond1=1,
2: emergency_distance_pedestrian_distance=1,
3: detection_pedestrian_tracking=1,
```

Figure 5.11: Solutions for the function *build_goal1_cond1()* containing AND operator.

```
[[1, 2, 3]]
Solution 1:
[1, 2, 3]
1: goal1_cond1=1,
2: emergency_distance_pedestrian_distance=1,
3: detection_pedestrian_tracking=1,
```

Figure 5.12: Solution presented after testing *build_goal1_cond1_True()* function.

```
[[-1, 2, -3], [-1, -2, 3]]
Solution 1:
[-1, 2, -3]
1: goal1_cond1=0,
2: emergency_distance_pedestrian_distance=1,
3: detection_pedestrian_tracking=0,


Solution 2:
[-1, -2, 3]
1: goal1_cond1=0,
2: emergency_distance_pedestrian_distance=0,
3: detection_pedestrian_tracking=1,
```

Figure 5.13: Solutions presented after testing *build_goal1_cond1_False()* function.

The numbers in the tables of the two Figures 5.12 and 5.13 represent the number of variables which in our case is 3 variables in Figure 5.11. Each variable can be true or false and is indicated by the sign in front of the variable number. If it has a negative sign, it means it is false or null. Otherwise, it is positive. If all the variables contained in the rule are true, it is possible to eliminate this solution because it does not make sense to have only all positive or negative solutions. If all the variables are false, they are automatically eliminated for the SAT problem.

5.6.2 Testing consistency of sequential and parallel rules with each other

To prove that two rules are inconsistent, we first test rules with their alerts by creating *build_goal1_cond1_Alert()* function wich takes the alert into account in the inconsistency test, and force it to be true with the function *build_goal1_cond1_Alert_True()* with one solution seen in Figure 5.14. Specifications or rules can have no solution

```
[[1, 2, 3, 4]]
Solution 1:
[1, 2, 3, 4]
1: accelerateEgo_cond1=1,
2: safe_distance_front_car_distance=1,
3: accelerateEgo_cond1_Alert=1,
4: acceleration=1,
```

Figure 5.14: Solution presented for the AND operator in **goal1_cond1** adding one variable for the acceleration alert.

or even one solution where it is always true/false. Experts can write parallel rules that sometimes they do not pay attention to the inconsistencies and must have priority, and can also write sequential rules that can be opposed. Consider the case of two rules R1 and R2 as shown in the table 5.8. Four combination cases exist: Safety expert has two ways of defining rules as seen in equation 5.6.

Table 5.8: Combination cases for the two rules.

Case	R1	R2	Explanation	Sequential	Parallel
1	0	0	Rule 1 and Rule 2 are false	0	0
2	0	1	Rule 1 is false, Rule 2 is true	1	1
3	1	0	Rule 1 is true, Rule 2 is false	1	1
4	1	1	Rule 1 and Rule 2 are true	0	1

For the sequential equation, it is the sum of [(R1∧¬R2)∨(¬R1∧R2)], and for the parallel (R2 ∨ R1). We eliminate as mentioned previously the solution when all rules are false.

$$Define(R1, R2) \begin{cases} Sequential\ definition, & (\text{R1}\wedge\neg\text{R2})\vee(\neg\text{R1}\wedge\text{R2}) \\ Parallel\ definition, & (\ \text{R2}\ \vee\ \text{R1}) \end{cases} \tag{5.6}$$

The priority execution study falls under cases 2 and 3, note that in the first case, the two rules are not triggered and the last case cannot be true since one is called before the other. For parallel execution, we sometimes miss giving priority between rules triggered at the same time which causes inconsistency.

If the safety expert sets R1 and R2 to run in parallel, it is necessary to know if the defined properties are consistent. From the table 5.8, the 4th case is when both rules are set to true. This is a case where we can see possible inconsistencies

since they are triggered together, so the operator is an AND operator on R1 and R2. To test this, we apply AND test defined in the equation 5.8 to see solutions when both are triggered together and analyze the results.
For the priority goal type, let us consider that R1 has a priority over R2 as following:

$$\begin{aligned} &if(R1)\{ \\ &\quad raiseAlarm(AlertR1); \\ &\}elseif(R2)\{ \\ &\quad raiseAlarm(AlertR2); \\ &\} \end{aligned} \tag{5.7}$$

If we consider $R1 = a$ and $R2 = a\&b$, the second condition is never going to happen and we never have **Alert2** executed because a $\subset a\&b$. Sometimes, the safety engineer does not notice that a requirement can never be triggered. To test this inconsistency, we follow tests in equation 5.8. These tests are generated to check and verify inconsistencies in the rules. If R1 is executed before R2, we apply $\neg$R1$\vee$R2 which constitutes the 3rd case in table 5.8 where R1 is true and R2 is false. To see when R2 is positive, we must test $\neg$R2$\vee$R1, the inverse where we can visualize the solutions when R1 is executed and not R2. If the solutions make sense, the safety engineer should question his choice, modify the requirements or even add new specifications. If the expert defines rule R1 which is sequential with rule R2 or vice versa, that means we need to test priority on R1 or R2 respectively. This test ensures that a rule is triggered before the other, and there is no case where one of them takes precedence. Finally, if R1 implies R2 and R2 implies R1, they are identical. This last test deals with the complexity listed in the requirements.

$$test(R1, R2) \begin{cases} \neg\text{R2}\wedge\text{R1}, & \textit{when priority execution of R2 over R1} \\ \neg\text{R1}\wedge\text{R2}, & \textit{when priority execution of R1 over R2} \\ \text{R1} \wedge \text{R2}, & \textit{when parallel execution} \\ (\text{R1} => \text{R2}) \wedge (\text{R2} => \text{R1}), & \textit{when rules are identical} \end{cases} \tag{5.8}$$

Consider the example in Figure 5.15 where two goals are executed simultaneously. The AccelerateEgo goal has an acceleration behavior whenever the distance to the front car is safe. The DecelerateEgo goal slows down the EV when the distance requires an emergency maneuver with an imminent collision distance. As these rules are executed in parallel, we test the AND between these two rules. Remember that this is a simple example to see the automatically generated tests. Figure 5.16 shows the 15th solution presented when all variables are true. Figure 5.17 is the solution when these two rules are triggered in parallel and are both true. Variables 4 and 8 are contradictory since either can be true. We can see acceleration and deceleration states are true at the same time which

```
RuleBasedPlanner RBP{
    scene ^Scene
    GOAL accelerateEgo{
    GoalType Constraint
    WHEN{
            propertyType "EgoProperties.front_car_distance" is
            "EgoProperties.front_car_distance.safe_distance"

        action "libActions.accelerate"
        alert "LibAlerts.acceleration"
    }
        GOAL decelerateEgo{
            GoalType Constraint
            WHEN{
                propertyType "EgoProperties.front_car_distance" is
                "EgoProperties.front_car_distance.imminent_collision_distance"

        action "libActions.decelerate"
        alert "LibAlerts.deceleration"
            }
        }
    }
}
```

Figure 5.15: The AccelerateEgo and DecelerateEgo rules run in parallel.

```
Solution 15:
[1, 2, 3, 4, 5, 6, 7, 8, 9]
1: accelerateEgo_cond1=1,
2: safe_distance_front_car_distance=1,
3: accelerateEgo_cond1_Alert=1,
4: acceleration=1,
5: decelerateEgo_cond1=1,
6: imminent_collision_distance_front_car_distance=1,
7: decelerateEgo_cond1_Alert=1,
8: deceleration=1,
9: accelerateEgo_cond1_decelerateEgo_cond1_Alerts=1,
```

Figure 5.16: Results of a parallel test applied using the AND operator on the AccelerateEgo and DecelerateEgo rules.

is inconsistent. Variables 2 and 6 are also contradictory since we cannot have a safe distance and a collision imminent distance at the same time. Seeing a lot of states incoherent, the safety engineer needs to change the GoalType and set a priority. Then, a test is performed according to the formulas of the equation 5.8. Deceleration has a priority over acceleration, which makes more sense. The best solution is to reverse the two rules because the deceleration in the AD is critical for vehicle safety.

Consider the example in Figure 5.18 where two goals are defined in sequence (with a Priority *GoalType*). The AccelerateEgo goal in this case has an acceleration behavior whenever the distance to the front car does not exist. The

```
Solution 1:
[1, 2, 3, 4, 5, 6, 7, 8, 9]
1: accelerateEgo_cond1=1,
2: safe_distance_front_car_distance=1,
3: accelerateEgo_cond1_Alert=1,
4: acceleration=1,
5: decelerateEgo_cond1=1,
6: imminent_collision_distance_front_car_distance=1,
7: decelerateEgo_cond1_Alert=1,
8: deceleration=1,
9: accelerateEgo_cond1_decelerateEgo_cond1_Alerts=1,
```

Figure 5.17: Results of a parallel test forcing the AccelerateEgo and DecelerateEgo rules to be true.

MaintainEgoSpeed goal remains with the same speed whenever the distance to the front car does not exist and the system does not track it. We execute the

```
RuleBasedPlanner RBP{
    scene ^Scene
    GOAL accelerateEgo{
    GoalType Priority
    WHEN{
            propertyType "EgoProperties.front_car_distance" is
            "EgoProperties.front_car_distance.not_exist"

        action "libActions.perform_acceleration"
        alert "LibAlerts.acceleration"
    }
        GOAL maintainEgoSpeed{
            GoalType Constraint
            WHEN{
                AND(propertyType "EgoProperties.front_car_distance" is
            "EgoProperties.front_car_distance.not_exist"
            ,
                propertyType "EgoProperties.front_car_tracking" is
                "EgoProperties.front_car_distance.not_exist"
                )
        action "libActions.maintain_speed"
        alert "LibAlerts.maintain_speed"
            }
        }
    }
}
```

Figure 5.18: The AccelerateEgo and MaintainEgoSpeed rules executed sequentially.

priority test with MaintainEgoSpeed $\vee$ $\neg$AccelerateEgo to see if there is a solution where it presents an inconsistency. Figure 5.19 shows the solution where maintaining speed and accelerating are both true, so **maintain_speed** is never executed since **not_exist_front_car_distance** is a common variable used in both goals, which means goal1 does not let the system pass to goal2 that is included in goal1's condition. This is how experts check that the priority is not well written and they need to change the goals or even maybe the type of execution.

```
[[1, 2, 3, 4, 5, 6, 7, 8]]
Solution 1:
[1, 2, 3, 4, 5, 6, 7, 8]
1: accelerateEgo_cond1=1,
2: not_exist_front_car_distance=1,
3: accelerateEgo_cond1_Alert=1,
4: acceleration=1,
5: maintainEgoSpeed_cond1=1,
6: not_exist_front_car_tracking=1,
7: maintainEgoSpeed_cond1_Alert=1,
8: maintain_speed=1,
```

Figure 5.19: Result of a priority test of MaintainEgoSpeed and AccelerateEgo functions to check inconsistency.

5.6.3 Testing consistency of the whole system

To test consistency of the whole rule-based planner, we need to parse the solutions to the ruleset. Generating a function that gives all the solutions is a way to deal with the consistency of the whole system. The rules that are executed sequentially are translated to this form:

$$\textbf{G1 priority over G2} = G1 \vee ((\neg G1) \wedge G2) \tag{5.9}$$

The rules that are executed in parallel are translated to this form:

$$\textbf{G1 parallel with G2} = G1 \vee G2 \tag{5.10}$$

$$\begin{aligned} \textbf{with } G1 &= R1 \wedge Alert1 \\ G2 &= R2 \wedge Alert2 \end{aligned} \tag{5.11}$$

The coherence of the system is the coherence of all the rules with their behaviors. Let us take the example in Figure 5.15 with a priority execution. Figure 5.20 shows the translation to study the coherence and the solutions of the system with all the rules. The *coherence_decelerateEgo()* function takes into account the negation of *coherence_accelerateEgo()*, considering or not the behavior triggered (the commented lines are with behaviors). We then create a function *coherence_all_goals_behaviors()* that uses an OR logic operator for all goals coherences. The system solutions are equal to coherence_all_goals_behaviors and coherence_all_properties as seen in the following generic equation 5.12:

$$\textbf{System\_solution} = coherence\_all\_goals\_behaviors \wedge coherence\_all\_properties \tag{5.12}$$

```
static public void coherence_all_goals_behaviors(IBooleanSpecification spec){
    coherence_accelerateEgo(spec);
    coherence_maintainEgoSpeed(spec);
    spec.or("coherence_all_goals_behaviors",
            "coherence_accelerateEgo", "coherence_maintainEgoSpeed");
}

static public void coherence_accelerateEgo(IBooleanSpecification spec){
    //Sat4jRulesConsistency.build_accelerateEgo_cond1_Alert(spec);
    Sat4jRules.build_accelerateEgo_cond1(spec);
    spec.or("coherence_accelerateEgo", "accelerateEgo_cond1");
}

static public void coherence_accelerateEgo_FALSE(IBooleanSpecification spec){
    //Sat4jRulesConsistency.build_accelerateEgo_cond1_Alert_False(spec);
    Sat4jRulesConsistency.build_accelerateEgo_cond1_False(spec);
}

static public void coherence_maintainEgoSpeed(IBooleanSpecification spec){
    coherence_accelerateEgo_FALSE(spec);
    //Sat4jRulesConsistency.build_maintainEgoSpeed_cond1_Alert(spec);
    Sat4jRules.build_maintainEgoSpeed_cond1(spec);
    spec.or("coherence_maintainEgoSpeed", "maintainEgoSpeed_cond1");
}
```

Figure 5.20: Code for translating sequential execution of the AccelerateEgo and DecelerateEgo rules to test the system coherency.

This generic equation is characterized in Figure 5.21 where for the example in Figure 5.15, **coherence_all_properties** is expressed as follow:

$$\begin{aligned} &\textbf{coherence\_all\_properties} \\ &= \textbf{coherence\_safe\_distance} \vee \textbf{coherence\_imminent\_collision\_distance} \\ &= (safe\_distance \wedge \neg imminent\_collision\_distance) \vee \\ &\quad (\neg safe\_distance \wedge imminent\_collision\_distance) \end{aligned} \tag{5.13}$$

The **system_solution** formula is generated by EPSAAV tool to help the user verify solutions for his rule-based planner. One of the states for each property should be activated and not both. We can not have a safe distance and an imminent collision distance at the same time. We dig into more examples and

```
static public void system_solution(IBooleanSpecification spec){
    coherence_all_properties(spec);
    coherence_all_goals_behaviors(spec);
    spec.and("system_solution", "coherence_all_properties", "coherence_all_goals_behaviors");
}
```

Figure 5.21: Code for translating sequential execution of the AccelerateEgo and DecelerateEgo rules to test the system coherency.

analysis in Section 6.7 and 6.8.

5.7 Conclusion

In this Chapter, we have discussed the necessity of using a constraint solver. We restrict ourself to a SAT solver (SAT4J) as it is sufficient to prove our point and has enough expressiveness for the kind of language used in safety requirements. However, note that encoding the enumeration types may lead to very inefficient encoding that did not justify by itself to use an SMT solver instead. To detect conflicts, depending on the structure of the rules we use the SAT solver to prove the system is satifiable or valid or falsifiable. Our framework is in charge of generating all the codes to check inconsistencies. The objective is to arrive at a solid and complete system that helps the engineer to integrate his safety requirements knowing that the inconsistencies will be dealt with.
We presented the three implementation phases to deploy the SAT solver. First, a Java generation is performed according to the safety rules defined by the expert. This generation follows a systematic encoding into Boolean formulas based on the structure of the rules. A set of Boolean operators are combined into these Boolean logic formulas that are fed to the SAT solver. Obviously, a lot of effort needs to be put into the infrastructure to code the generator to support these features. However, it reduces the development time for the safety engineer who may not be an expert in programming. In a second step, inconsistency tests on the rules and the system are carried out. We detail the generation Java code to study the inconsistencies between the rules. Java code is represented in three files: Sat4jRules.java, Sat4jRulesConsistency.java and Sat4jSystemConsistency.java. We dig deeper into the application of safety requirements for a Renault use case in the Chapter 6 to assess the usability of our approach and of our verification engine on an industrial case study.

Chapter 6

Application of EPSAAV Approach to a Renault Use Case

6.1 Introduction

The EPSAAV language in Chapter 3 and the generated monitor and verification engine with the environments presented in Chapter 4 and 5 are defined by its abstract syntax which captures all Safety domain concepts and its semantics and concrete syntax close to the safety engineer terminology. By interacting with a provided concrete syntax, the safety engineer is guided to provide a syntactically and semantically correct and complete instance model, which includes the safety goals or rules, expected properties, as well the targeted scenes with associated safety behaviors (alerts and actions). **EPSAAV specifier** constitutes the definition and configuration of these rules and environment by using a formalization technique, and provides the internal grouping structure. The produced specification is considered as input for the **EPSAAV generator** which enables generation of documents, verification tools, and monitors. Both **EPSAAV specifier** and **generator** constitute the thesis contributions. Developers can assess, and validate the generation process, while safety engineers can directly use generated documents and tools.
The impact of EPSAAV is interesting from Renault's point of view, which provided a testbed for validating the language. With many organizations developing their languages or hiring companies to develop such languages for them, our DSML can aid them in conducting safety evaluations.
In this Chapter, we present a feasibility study that combines verification and validation. First, we present the traditional scenario in safety engineering and the problems it may face, the workflow using EPSAAV tool, and the improvements of the proposed DSML in Section 6.2. In our use case, we refer to the Safety domain, note that the approach can be applied to other domains such as security. We present the concrete implementation of the domain concepts (introduced

also in Chapter 3), and introduce an industrial case study describing scenarios in Section 6.3, and an instantiation of EPSAAV in Section 6.4. Finally, we carry out a pilot empirical evaluation of the implemented tool in Sections 6.5, 6.6 and 6.7. This illustrates the possibility of capturing all relevant information to perform a DSML assessment and how it can be used in the analysis of the results and modification of the initial requirements following the SAVI process in Section 6.8.

6.2 Safety engineering workflow

In automotive research companies, many techniques are used to evaluate safety. One of these techniques is scenario planning which allows decision-makers to identify ranges of potential outcomes and estimated impacts. It evaluates responses and manages positive and negative possibilities while presenting significant risks. When a worst-case event occurs, scenario planning documents add tremendous value by playing on multiple outcomes and listing immediate steps to contain the damage [134]. Scenario planning can document actions and emergency decisions that constitute a clearer picture of key drivers for AV's growth and the potential impact of future events. In the automotive industry, two main actors exist to validate safety. First, a safety engineer is a professional who applies the principles for the design and evaluation of unsafe scenarios. By safety engineer, we refer in this thesis to the safety system engineer who has the task of defining software requirements related to functional and safety. Second, developers are professionals who could support monitoring safety assessment while being responsible for running code on various components of AV. Safety engineering is a team activity to evaluate and specify safety requirements.
Safety engineers make sure AVs are safe by monitoring the work environment and inspecting hazards and safety violations. They recommend safety features in new processes and evaluate plans for new specifications. They assure that a life-critical system behaves as needed even when features fail. All experts use safety standards and evaluate unsafe scenarios to grant each one a *human-readable document*. While safety analysis techniques rely on the expertise of the engineer, they depend also on the types of techniques they are following. Traditional techniques are hard to derive relationships between causes and consequences. It is always good to spot details and explanations of violations. The two most common fault techniques used in Renault are Failure mode and effects analysis (FMEA) [135] and Fault tree analysis (FTA) [136] which are the ways of finding problems and assessing probabilistic risk. In Renault Group, automotive safety engineers used traditional methods to provide acceptable levels of safety, which workflow we discuss in the Section 6.2.1. Using the EPSAAV, the engineers are empowered with a new workflow represented in the Section 6.2.2. We discuss the benefits of the new approach in Section 6.2.3.

6.2.1 Traditional workflow

In the traditional workflow, safety engineer defines requirements in a *human-readable document* (e.g. pdf document), which is then sent to development team (see in fig. 6.1). A Software Architecture maps the full set of software requirements to software components and defines interfaces for each. In this workflow, we link directly safety engineers (or safety SW engineers) with the developers. Developers need to develop solutions and deploy the requirements on all the components. This process can take time. The developers then implement *monitor*

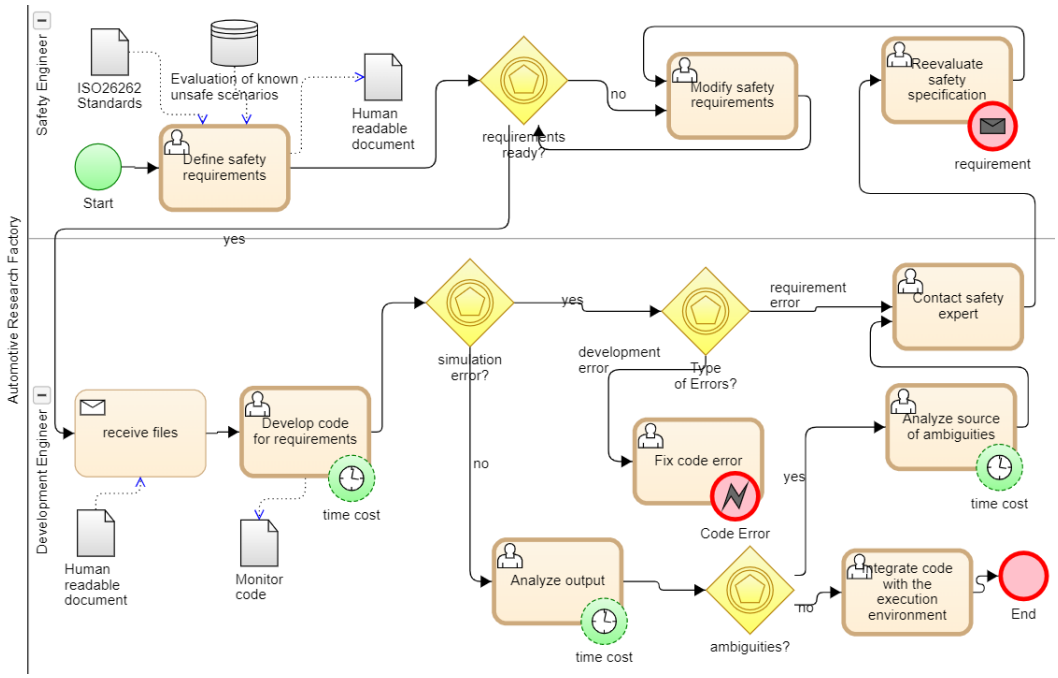

Figure 6.1: Traditional workflow in Renault Group.

code which may constitute simulation errors. If a simulation error exists, they start looking for the type of error. Errors can be related to development or requirements. If it's a development failure, they correct the problem and starts analyzing if the output meets the expert's specifications. If all goes well, they integrate the monitor into the run-time environment. If not, they analyze the source of the error such as missing conditions in the rule. They need to understand the requirements to prove that their actions are coherent. For instance, if they wait for a deceleration alert at a certain step, they consider it as an ambiguity in the definition of the rules that pushes them to contact the safety engineers to reassess specifications. If a requirement error is detected, the safety engineers edit the requirements and send them back to the developers.

6.2.2 Workflow using EPSAAV language

Fig. 6.2 shows the new workflow using DSML support. The safety experts define safety requirements using the formal syntax close to their domain concepts. They then obtain three different types of generated outputs from the tool:

1. A *human-readable document* that allows them to validate the correctness of their specifications,
2. a *verification engine* that enables the experts to check and analyze the inconsistencies of the rule using a **solver**, and
3. a *monitor code* that is inserted by the developers into the **debugger**.

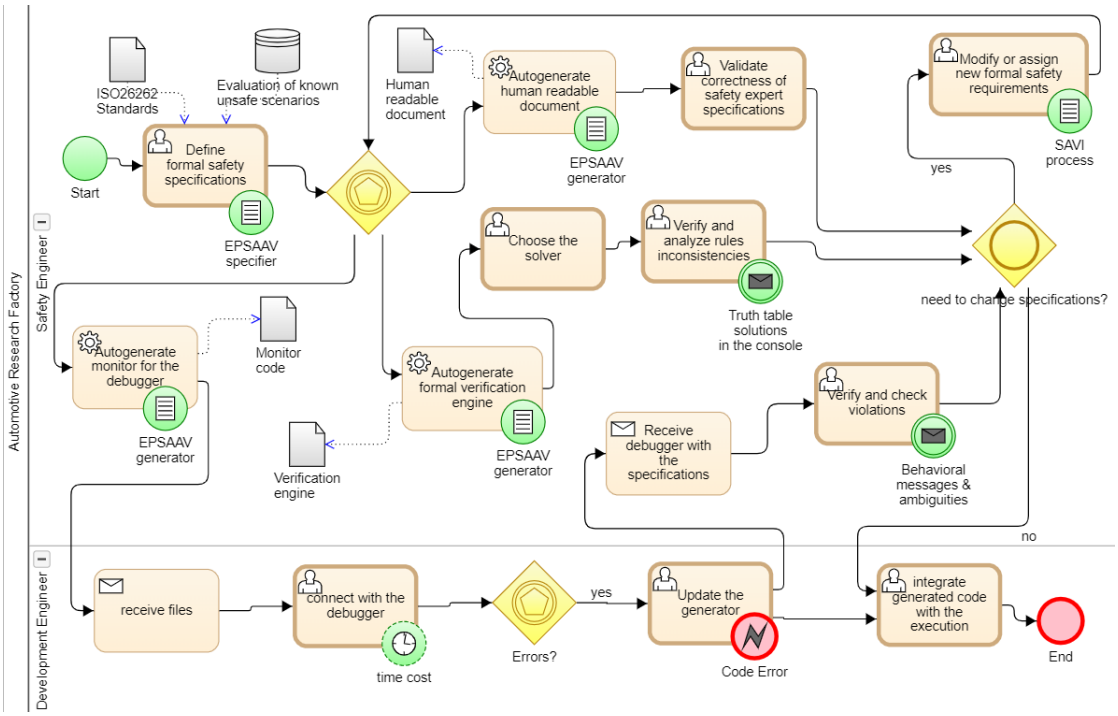

Figure 6.2: Workflow using EPSAAV language.

The developers take the code from the monitor and connect it to the debugger linking all input data with simulated or real data output. They then run the code and update the generator if there are any errors. They hand over the modified platform to the experts who check and verify the violations by visualizing all ambiguities with behavioral messages. After receiving all the implementations from the developers, the safety experts then study whether there is a need to modify the specifications. If so, they go through a Safety Analysis of Violations and Inconsistencies (SAVI) process that we introduce later in 6.8. This process helps them assign new formal safety requirements and automatically generate the right document and monitor to assess safety.

6.2.3 Workflow improvements and EPSAAV benefits

Traditional workflow presents many disadvantages for the automotive industry. First, the results of traditional risk analysis are difficult to apply directly to modern software design, especially since the code may have errors when translated. The safety expert does not have the means to verify the correctness of requirements directly, as there is no guide to address potential vulnerabilities and threats at the component level. Rules may contain inconsistencies, be repeated multiple times, or even have the wrong execution type of goal (executed in parallel instead of sequentially, vice versa).
Second, development engineers can also misinterpret specifications from safety experts and produce conceptual errors when coding, which is difficult to be captured by a safety engineer. As the system becomes more complex and contains more rules, developers need much more time to develop and memorize what was done.
Third, developers are sometimes tasked with analyzing the output and source of ambiguities, and due to their lack of safety experts, it is difficult for them to prove the correctness of safety requirements so they contact the expert. As this can take a lot of time, sometimes the developer skips the analysis and deploy directly the code to the execution environment. All of these threats can cause costs in communication and development problems whenever a change is required, as there is no automatic link between the fields of safety and development.

With the support of **EPSAAV specifier**, we provide the safety expert with a tool for formalizing their needs, which helps them follow a single format to avoid misunderstandings with other engineers. The correctness of rules specification is syntactically and semantically validated and proven, and the engineers visualize better their inputs, unlike the first workflow where they do not follow a specific pattern to configure his inputs.
A second advantage is the concept of generation of a document, a monitor, and verification engine which saves development time and ensures consistency of rules by using a solver and a simulation environment, which as result reduces costs and improves correctness. The developer no longer needs to create the *monitor code* by hand for each specification. The **EPSAAV generator** automatically generates the monitor giving a single task to the developer to connect it to the debugger.
Another advantage of portability allows easy adaptation of the code to several languages instead of concentrating on a single generator. Developers can implement with ease a new code generator in the case of a system change, or extend it and test the functional correctness by reusing the existing implementation. In other words, we can generate many languages by modifying the existing code in Xtend projects, created to automatically generate human-readable documents and files. In the traditional workflow, if the industry decides to switch programming languages, developers have to redo all the work from scratch, leaving more

room for error and more work to be done. Regarding the validation of specifications and the checking of inconsistencies in its defined rules, safety engineers can visualize violations and investigate ambiguities by using the verification engine, without the need to wait for the development process to be completed in order to change its rules. He touches the real world and bridges the safety realm without needing to spend a lot of time in the development process.
Finally, the safety engineers are provided with the same *human-readable document* which was used in traditional flow for their specification, so they can conceptually validate what was specified in the platform and use it for communication with other safety engineers. With the monitors and engines generated in the new process, developer costs are reduced and the safety engineer takes full responsibility for analyzing their requirements using monitors provided by **EPSAAV specifiers** and **generators**.

6.3 From *AREA2* to *Area2Spec* using EPSAAV language

We dedicate this Section to explain the scenarios developed in a Renault project that assessed known hazardous scenarios, named *AREA2* for known unsafe scenarios that is represented in Chapter 2 in Section 2.3.2. The *AREA2* was developed in collaboration with the consulting company Tecris to help AD systems for level 3 to better assess dangerous scenarios. The measures and risks are formalized in this thesis, in a pdf file called *Area2Spec*, with the help of an IVEX project in Renault Group where they developed a co-pilot to assess safety. We detail this transformation and the advances favored by the proposed approach.

6.3.1 Evaluate unsafe known scenarios

The *AREA2* document was developed to define the SOTIF requirements. It was created to reduce the barriers between the safety field and development engineering. The main goal for developers is to program AD using the *AREA2* document. It identifies risks and assesses performance limits. *AREA2* takes into account failures to specify functional and system AD mode in a SOTIF HARA [137]. SOTIF can be seen as an extension of the Functional Safety (FuSa) [138] approach described in the V-shaped ISO 26262 standard designed to address the challenges of automated driving functions but with augmented components. The objective of *AREA2* is to determine the greatest number of use cases for the system and to verify that Renault can handle all scenarios in critical situations. The *AREA2* document uses several strategies and expertise from the System, Fusion, Validation, and Security teams to verify and validate the use cases. Methodologies for doing validation and verification consist of testing and validating that any scenario can be handled by the EV. These tests can be simulations to check

that any scenario with a significant variation in the initial parameters does not lead to an accident. They can also be in the field to apply it to specific environmental conditions to validate the capability of perception sensors and verify a scenario that has not been successfully run in simulation. The advantage of the proposed approach is that it can be applied to a real or simulated environment. Thus, both tests can be performed and analyzed using the **EPSAAV specifier** and **generators**. Another test that can be done is on the open road to test the EV against real-world environmental conditions. When all rules are well specified and approved by safety experts and developers, the developed tool could be deployed on these open roads to assess safety during known dangerous scenarios. The triggered actions are useful for planning, and the alerts are essential for the driver to know what is happening around. *AREA2* serves as a complement to test teams. It describes all the parameters required to prepare the tests and estimate the combinations.
AREA2 document presents 13 scenarios evaluating known unsafe situations. In our case study, we evaluate safety in five scenarios that constitute longitudinal and lateral controls.

6.3.1.1 Risk of a frontal collision with the PV in deceleration scenario

The scenario is represented in Figure 6.3 where EV is on the highway in a traffic jam. There is a PV in front of the EV and is making a deceleration regarding if it is a strong braking or nominal deceleration.

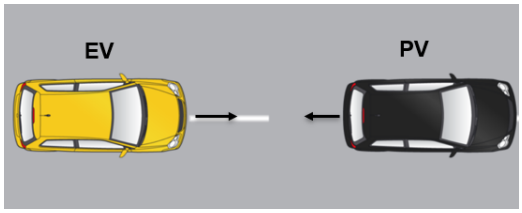

Figure 6.3: Scenario of a PV decelerating.

The risk is to have a front collision with the PV.

The measures to avoid the collision with the PV are the following:

1. The system shall detect the PV that is decelerating.
2. It should maintain a safe distance by braking.
3. In case of an accident, a report should be done for the Delta-V. Delta V specifically refers to the change in velocity between pre-collision and post-collision trajectories of a vehicle.

The parameters that are applied on this and all following scenarios consist of values for deceleration types. These values were given by Renault Safety Team that are then used for safety evaluation.

- ACC braking when there is an ACC distance of 1.2 seconds. In this case, it is not a safe distance anymore but less.
- Strong braking when there is a strong braking distance of 0.8 seconds.
- Emergency maneuver is the worst case and shall be triggered when the system detects an imminent collision with the others. So it is less than 0.8 seconds and usually, it requires maximum braking.

The safety distance is the 2 seconds rule that is used in Renault and is a rule of thumb by which a driver may maintain a safe trailing distance at any speed [139].

6.3.1.2 Risk of rear collision with the FV due to a false recognition scenario

The scenario is represented in Figure 6.4 where EV in on the highway in traffic jam. The EV is between the Following Vehicle (FV) and the PV. The EV starts creating a false recognition. By false recognition, we can mention:

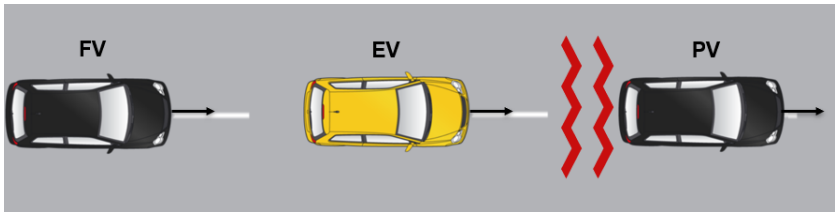

Figure 6.4: Scenario EV between the FV and the PV and an interference occurs.

- Interferences that may be caused by entering a tunnel,
- or even a wrong detection of objects.

The risk is to have a rear collision with the FV.

The measures to avoid the collision with the FV are the following:

1. The system shall detect the obstacle on the road in front of the EV.

2. It should make a deceleration and confirm the obstacle.

   a- During the confirmation phase, the EV executes an ACC braking,

   b- in case of positive confirmation, the EV executes a necessary value of braking,

   c- in case of negative confirmation, the EV should not apply a brake.

3. The system shall apply emergency braking in case of an imminent collision.

4. In case of an accident, a report should be done for the Delta-V.

6.3.1.3 Risk of a side collision with SV due to missing lane detection scenario

The scenario is represented in Figure 6.5 where EV is on the highway in a traffic jam. The EV is between the FV and the PV, and there are other users such as Straddling Vehicle (SV) or guardrail. The EV starts creating a disturbance preventing line or lane detection.

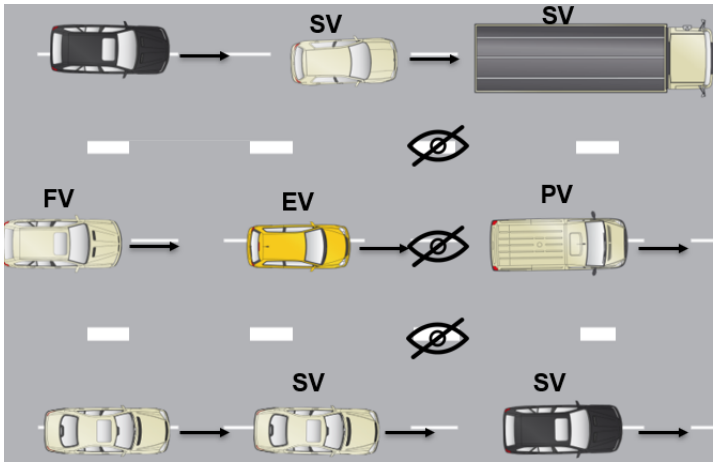

Figure 6.5: Scenario of an EV between the FV and the PV and next to SVs, and a line/lane disturbance occurs.

The risk is to have a lateral collision with an SV.

The measures to avoid a collision with the SV are the following:

1. The system shall activate the lighting or the wiping system sending to the driver an alert.

2. In case of loss line the system shall maintain the vehicle in the lane using the remaining information about the PV.

3. In case of loss of PV, the system shall maintain the vehicle in the lane using the remaining information about SV.

4. If the line disappears more than a threshold in seconds, the EOP1 shall be triggered. The EOP1 requires the driver to take his hands on the wheel and should be prepared to take over.

5. If the PV disappears more than a threshold in seconds, the EOP1 shall be triggered.

6. The worst case is when the system detects an imminent collision with SV, the emergency maneuver shall be triggered.

6.3.1.4 Risk of a side collision with the SV due to poor infrastructure scenario

The scenario is represented in Figure 6.6 where EV is on the highway in a traffic jam. On the left and right sides of EV, there are other road users such as SV or guardrail. Disturbances in infrastructures prevent correct detection of the lane such as an old-line visible or missing road markings in many situations such as a cut-in case.

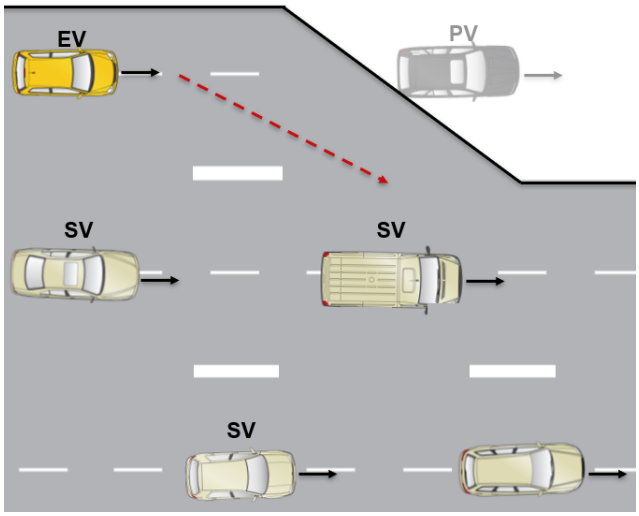

Figure 6.6: Scenario of an EV making a cut-in and does not detect the guardrail due to an invisible old line.

The risk is to have a lateral collision with the SV.

The measures to avoid a collision with the SV are the following:

1. The system shall detect the incoherence between sensors that cause loss of line/lane detection.
2. The system shall maintain the vehicle in the lane using the remaining information of PV, SV, and lines.
3. If the line disappears more than a threshold in seconds, an EOP1 is triggered.
4. The worst case is when the system detects an imminent collision with SV, the emergency maneuver shall be triggered.

6.3.1.5 Risk of a side collision with SV swerving into Ego lane scenario

The scenario is represented in Figure 6.7 where EV is on the highway in a traffic jam. The EV is following the PV. An SV is straddling over the Ego lane.

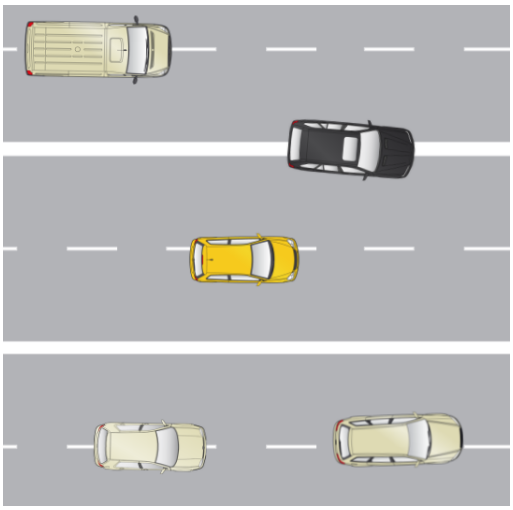

Figure 6.7: Scenario of an SV that is taking EV lane and straddling.

The risk is to have a lateral collision with the SV.

The measures to avoid a collision with the SV are the following:

1. The system shall regulate the safe distances with SV and stay behind.
2. The Ego shall continue to follow the same line and the trajectory of PV.
3. If the SV continues to straddle after a threshold, an EOP1 shall be triggered.
4. The worst case is when the system detects an imminent collision with SV, the emergency maneuver shall be triggered.

6.3.2 IVEX co-pilot and *Area2Spec* document

The Renault group has collaborated with the IVEX team to apply their co-pilot which analyzes the safety of AVs [140]. IVEX's vision is to reduce traffic casualties and make AVs and semi-AVs safe [141]. Their objective was to provide a safety model to focus on longitudinal and lateral controls and to study real and simulated datasets provided by Renault. The overview is based on IVEX's elaborated work and weekly discussions with Renault and IVEX's analysis activities. The scope of activities focused on one-way freeway situations, with particular attention to the longitudinal safety distance. Work started from the IVEX safety model which includes an improved variant of the RSS safety rules [12] for the longitudinal safety distance. Improvements include runtime adaptation of particular settings that retain safety properties but allow for more realistic and reasonable ego car behavior.
IVEX has used the previous scenarios presented to check safety using their co-pilot. This latter is used to continuously verify the planned trajectory and assess the operational safety of the trajectory. An important goal of this collaboration is that we formalized the safety test measures on previous scenarios. This document is called *Area2Spec* and is used for their safety assessment on multiple Renault real data scenarios. This transformation used a logical expressiveness for the rules and inspired our work. The formalized rule proved that the approach of testing requirements in a logical expressiveness is a solution to evaluate safety. Renault presented the *AREA2* five goals containing the measures and risks. With the help of IVEX, we introduced a document called *Area2Spec* that contains all the formal rules of the *AREA2* document.

Operational Design Domain (ODD)

Safety rules considered in *Area2Spec* were initially structured and expressed in human text language. Situations covered for Level 3 autonomy were essentially in a traffic jam, at low speed, and various weather and light conditions as well as infrastructure and slopes. Organized in *Area2Spec*, the rules provide information about the situation is verified, the collision is avoided and specific behavior to consider. Since all scenarios are in a traffic jam that must be the operating case, known as the Operating Design Domain or ODD, we created a property **traffic_jam** that has **true** and **false** states. We note that the formal constructions provided for EV and PV are equally applicable to FV. This would however be flagged as impractical by the **EPSAAV generator**. The verification engine and the monitor code imply having priorities. Indeed, a priority is linked to levels of responsibility according to the highway code: EV can only be held responsible for blowing the front car via a longitudinal maneuver. Therefore, EV instructions should be specified as having priority over FV. This is done by changing the Goal Type implemented in the **EPSAAV specifier** to priority for the corresponding

instruction block.
Another aspect that IVEX and our work have highlighted is related to perception imperfections. It consists of the potential loss of traces of objects over time. With existing information from sensors used in vehicles, the driver assistance system is subject to loss of object tracking due to inaccuracies in location and trajectory parameter estimates. This causes object identifiers to disappear and new identifiers to be generated, potentially for the same objects. Safety rules must deal with such a case in order to decide the validity of issuing a new behavior. The corresponding **tracking** state values are then expressed in the formalization process.

6.3.2.1 Formalization of frontal collision with the PV in deceleration risk

Formal expressions (*FEs*) are addressed for each measure or requirement (*R*) in scenario (*S1*) presented in Section 6.3.1.1 as following:

1. *S1R1*= the system shall detect the PV that is decelerating.
   S1FE1= Introducing **front_car_distance** property.

2. *S1R2*= it should maintain a safety distance by braking.
   We introduce five states for **front_car_distance** that can be **not_exist**, **safe_distance**, **acc_distance**, **strong_braking_distance**, and **imminent_collision_distance**.
   We also introduce actions such as **emergency_maneuver**, **brake_strong**, and **brake_acc**.

   - *S1FE2_1*= when **front_car_distance** is **imminent_collision_distance** then goal is executing **emergency_maneuver**;
   - *S1FE2_2*= when **front_car_distance** is **strong_braking_distance** then goal is executing **brake_strong**;
   - *S1FE2_3*= when **front_car_distance** is **acc_distance** then goal is executing **brake_acc**;

3. *S1R3*= in case of an accident, a report should be done for the Delta V. Delta V specifically refers to the change in velocity between pre-collision and post-collision trajectories of a vehicle.
   S1FE3= when **front_car_distance** is **imminent_collision_distance** then goal is executing **report_longitudinal_delta_V**.

6.3.2.2 Formalization of rear collision with the FV due to a false recognition risk

Formal expressions (*FEs*) are addressed for each requirement (*R*) in scenario (*S2*) presented in Section 6.3.1.2 as following:

1. *S2R1*= the system shall detect the obstacle on the road in front of the EV. *S2FE1*= same as *S1FE1* requirement.

2. *S2R2*= it should make a deceleration and confirm the obstacle.

   - *S2R2_a*= during the confirmation phase, the EV executes an ACC braking,
     - *S2FE2_a_1*= introducing of **front_car_tracking** property with five states: **not_exist**, **not_confirmed**, **stable_tracking**, **disappeared_less_than_t1**, and **disappeared_more_than_t1** .
     - *S2FE2_a_2*= when **front_car_tracking** is **not_confirmed** then goal is executing **brake_acc**;
   - *S2R2_b*= in case of positive confirmation, the EV executes a necessary value of braking,
     S2R2_b is formalized previously in Section 6.3.2.1 in *S1FE2*.
   - *S2R2_c*= in case of negative confirmation, the EV should not apply braking.
     We force the system not to produce alerts and actions giving the priority to the **imminent_collision_distance** for the PV.
     - *S2FE2_c_1*= introducing of **rear_car_distance** property with three states: **not_exist**, **safe_distance**, and **emergency_distance**.
     - *S2FE2_c_2*= when **rear_car_distance** is **emergency_distance** then goal is executing **no_action_needed**.

3. *S2R3*= the system shall apply emergency braking in case of an imminent collision. *S2R3* is formalized previously in Section 6.3.2.1 in *S1FE2*.

4. *S2R4*= in case of an accident, a report should be done for the Delta V. *S2R4* is formalized in Section 6.3.2.1 in *S1FE3*.

6.3.2.3 Formalization of side collision with SV due to missing lane detection risk

Formal expressions (*FEs*) are addressed for each requirement (*R*) in scenario (*S3*) presented in Section 6.3.1.3 as following:

1. *S3R1*= the system shall activate the lighting or the wiping system sending to the driver an alert.
   This requirement is not the main concern for the thesis, this is why we did not include any activation of hardware systems.

2. *S3R2*= in case of loss line the system shall maintain the vehicle in the lane using remaining information about the PV.
   S3FE2= introducing of **line_detection** property with a state for the missing detection **no_detection_in_more_than_t6**, another state for **no_detection_in_less_than_t6**, and one **stable** state.

3. *S3R3*= in case of loss of PV, the system shall maintain the vehicle in the lane using remaining information about SV.
   S3FE3= introducing of **stable_control** property with **true** and **false** states.

4. *S3R4*= if the line disappears more than a threshold in seconds, the EOP1 shall be triggered. The EOP1 requires the driver to take his hands on the wheel and should be prepared to take over.
   S3FE4= when **line_detection** is **no_detection_in_more_than_t6** then goal is executing **emergency_operation_1**.
   We introduced the **emergency_operation_1** in the action library.

5. *S3R5*= if the PV disappears more than a threshold in seconds, the EOP1 shall be triggered.
   S3FE5= when **front_car_tracking** is **disappeared_more_than_t1** then goal is executing **emergency_operation_1**.

6. *S3R6*= the worst case is when the system detects an imminent collision with SV, the emergency maneuver shall be triggered.
   The formalization of this requirement is defined previously in Section 6.3.2.1 in *S1FE3*.

6.3.2.4 Formalization of a side collision with the SV due to poor infrastructure risk

Formal expressions (*FEs*) are addressed for each requirement (*R*) in the scenario *S4* presented in Section 6.3.1.4. All of them are formalized previously for the PV, and are applied to SV replacing the properties **front_car_distance** and **front_car_tracking**, with **straddling_car_distance** and **straddling_car_tracking** respectively.
Straddling_car_distance property has five states: that can be **not_exist**, **safe_distance**, **acc_distance**, **strong_braking_distance**, and **imminent_collision_distance**.
Straddling_car_tracking property has three states: **not_exist**, **straddling_less_than_t7**, and **straddling_less_than_t7**.

1. *S4R1*= the system shall detect the incoherence between sensors that cause loss of line/lane detection.
   We introduced a **bug_notification** that helps to study the bug from sensor perception.

S4FE1= when **front_car_distance** is **not_exist** if and only if **front_car_tracking** is **not_exist** or disappeared then execute **bug_notification**. Same thing is applied for the **straddling_car_distance** to test if there is a perception error.

2. *S4R2*= the system shall maintain the vehicle in the lane using the remaining information of PV, SV, and lines.
   Same formal pattern is used in Section 6.3.2.3 in *S3FE3*.

3. *S4R3*= if the line disappears more than a threshold in seconds, an EOP1 is triggered. Formalization for this requirement is done in Section 6.3.2.3 on *S3FE5*.

4. *S4R4*= the worst case is when the system detects an imminent collision with SV, the emergency maneuver shall be triggered.
   The formalization of this measure is defined previously in Section 6.3.2.1 in *S1FE3*.

6.3.2.5 Formalization of side collision with the SV swerving into Ego lane risk

Formal expressions (*FEs*) are addressed for each measure (*R*) for the scenario *S5* presented in Section 6.3.1.5, where *S5R1*, *S5R2* and *S5R4* are previously defined.

1. *S5R1*= the system shall regulate the safe distances with SV and stay behind.

2. *S5R2*= the Ego shall continue follow the same line and the trajectory of PV.

3. *S5R3*= if the SV continues to straddle after a threshold, an EOP1 shall be triggered.
   S5FE3= when **straddling_car_tracking** is **straddling_more_than_t7** then goal is execution **emergency_operation_1**.

4. *S5R4*= the worst case is when the system detects an imminent collision with SV, the emergency maneuver shall be triggered.

6.3.3 EPSAAV benefits over IVEX co-pilot

There are many drawbacks to using the IVEX Assessment Tool. The *Area2Spec* requirements that they used are predefined and therefore fixed. Once defined, we could not make any changes after their studies or add new requirements. Each modification or introduction of rules implies significant manual modifications. The safety engineers at Renault do not have access to the safety requirements,

so they cannot test their specifications using the co-driver. Their co-pilot is proprietary, the only purpose of the use is to display certain violations in specific scenarios. Another disadvantage is that they did not present a formalism to study the inconsistencies in each rule and between a pair of rules. What we want is to help the engineer to express their needs and modify their specifications to assess safety.
The proposed approach comes to the rescue by giving full access to the tool to edit existing specifications. It also gives all possible solutions for all rules defined using **EPSAAV generator**, so that the experts know where to apply changes if a solution contradicts what they expect. **EPSAAV generator** also gives him the ability to choose a specific simulator and solver and automatically generate monitors that help reduce development time and cost.

6.4 RBP and libraries instantiation for *Area2Spec* using EPSAAV specifier

We dedicate this Section to illustrating the EPSAAV conceptual framework with a case study about the *Area2Spec* document for safety requirements. All the files instantiated are represented in the Appendix A.1. EPSAAV was developed to lower the barriers between the safety experts. A safety expert may produce a safety rules document in which the rules may be duplicated or contradictory. Experts cannot successfully analyze everything if they do not have a common documentation platform and a tool where all the rules are inserted in one place. It was also conceived to subordinate the work of developers and instantly assess safety. The safety engineer has to specify the requirements using the **EPSAAV specifier**. As mentioned in Chapter 3, we created six files to create *Area2Spec* as following:

- A library for the alerts named *Area2Al.alerts* as seen in Figure 6.8. It is based on the grammar defined in Chapter 3 in Section 3.3.3.5. It contains all the alerts that we want to pass to the driver.

```
AlertLibrary Area2Alerts :
Alert "longitudinal_acceleration";
Alert "longitudinal_information_reporter";
Alert "lateral_information_reporter";
Alert "bug_notification";
Alert "eop1";
Alert "no_alert_needed";
```

Figure 6.8: Alert library for *Area2Spec* defined in *Area2Al.alerts*.

- A library for the actions named *Area2Ac.actions* containing all actions needed in Figure 6.9. It is based on the grammar defined in Chapter 3 in Section 3.3.3.5.

```
ActionLibrary Area2Actions :
Action "brake_acc";
Action "brake_strong";
Action "report_longitudinal_delta_V";
Action "report_lateral_delta_V";
Action "emergency_operation_1";
Action "emergency_maneuver";
Action "bug_notification";
Action "no_action_needed";
```

Figure 6.9: Action library for *Area2Spec* defined in *Area2Ac.actions*.

- A library for the object type named *Area2Obj.otp* (seen in Figure 6.10). It is based on the grammar defined in Chapter 3 in Section 3.3.3.3. This was not defined in *Area2Spec* to give power to the developer to attack the parameters that need to be taken into account. For the Ego, we want to define for each step the timestamp for the detected or missing lane. For the obstacle, we define an ID and counters for tracking or missing obstacle, and a parameter for Time To Collision (TTC).

```
ObjectTypeLibrary OTL version'1':

Ego {
    EgoPropertyLibrary EgoProperties;
    EgoParameterLibrary Ego :
        parameter Lane_detection Type float;
        parameter Lane_missing Type float;
}

"Obstacle"{
    ParameterLibrary Obstacle :
        parameter id Type int32;
        parameter counter_tracking Type float;
        parameter counter_not_tracking Type float;
        parameter ttc_min Type float;
}
```

Figure 6.10: Object type library containing parameters and refering to property library using EPSAAV specifier in *Area2Obj.otp*.

- A file for the scene named *Area2Sc.scene* containing all the capacity of perceiving objects. It is based on the grammar defined in Chapter 3 in Section 3.3.3.2. Each object refers to an ObjectType defined previously. In *Area2Spec*, we included PV, SV, and FV that have an Obstacle type, and the EV that is the Ego that we want to evaluate safety rules on. Figure 6.11 shows the input data for the scene.

- A library for the Ego properties named *Area2PEgo.prop* (seen in Figure 6.12). It is based on the grammar defined in Chapter 3 in Section 3.3.3.4. It contains all properties and states that we want to use to define the requirements. The safety engineer shall decide, if it is necessary, what is the threshold for each state. These thresholds depend on AEB braking system

```
scene Scene {
    ego ^Ego
    role Ego objectType "OTL.Ego"
    role "preceding_vehicle" objectType "OTL.Obstacle"
    role "straddling_vehicle" objectType "OTL.Obstacle"
    role "following_vehicle" objectType "OTL.Obstacle"
}
```

Figure 6.11: Scene for *Area2Spec* defined in *Area2Sc.scene*.

```
PropertyTypeLibrary EgoProperties
{   version '1'
    state "traffic_jam" CanBe "yes" "no"
    state "front_car_distance" CanBe "not_exist"
    "safe_distance" operator">=" value 2.0 unit "s"
    "acc_distance" operator">=" value 1.2 unit "s"
    "strong_braking_distance" operator">=" value 0.8 unit "s"
    "imminent_collision_distance"
    state "front_car_tracking" CanBe "not_confirmed"
    "not_exist" operator">=" value 0.33 unit "s"
    "disappeared_less_than_t1" operator"<" value 0.25 unit "s"
    "disappeared_more_than_t1" operator">=" value 0.25 unit "s"
    "stable_tracking" operator">" value 0.25 unit "s"
    state "line_detection" CanBe "stable"
    "no_detection_in_less_than_t6" operator"<" value 0.3 unit "s"
    "no_detection_in_more_than_t6" operator">" value 0.3 unit "s"
    state "straddling_car_distance" CanBe "not_exist"
    "safe_distance" operator"=" value 2.0 unit "s"
    "acc_distance" operator">=" value 1.2 unit "s"
    "strong_braking_distance" operator">=" value 0.8 unit "s"
    "imminent_collision_distance"
    state "straddling_car_tracking" CanBe "not_exist"
    "straddling_less_than_t7" operator"<" value 2.0 unit "s"
    "straddling_more_than_t7" operator">=" value 2.0 unit "s"
    state "stable_control" CanBe "stable" "not_stable"
    state "rear_car_distance" CanBe "not_exist"
    "safe_distance" operator"=" value 2.0 unit "s"
    "emergency_distance"
}
```

Figure 6.12: Ego property library for *Area2Spec* defined in *Area2PEgo.prop*.

specifications. It is important to note that distances are not expressed in meters, but in seconds. The **safe_distance** follows the two seconds rule. We can see in Figure 6.13 the differences between **front_car_distance** states. The most critical one is the **imminent_collision_distance**. Same thresholds are applied to **straddling_car_distance**. Other thresholds defined by the safety expert referring to the braking system specifications are:

  - For **front_car_tracking**, t1= 0.25 seconds and does not exist when it is more than 0.33 seconds.
  - For **line_detection**, t6= 0.3 seconds.
  - For **straddling_car_tracking**, t7= 2 seconds.

- A file for the rule-based planner named *Area2Spec.rbp*, containing all *FEs* expressed for all five scenarios defined in Section 6.3.2. It is based on the grammar defined in Chapter 3 in Section 3.3.3.1. After grouping all these formal requirements, we obtained four rules defined as goals as seen in

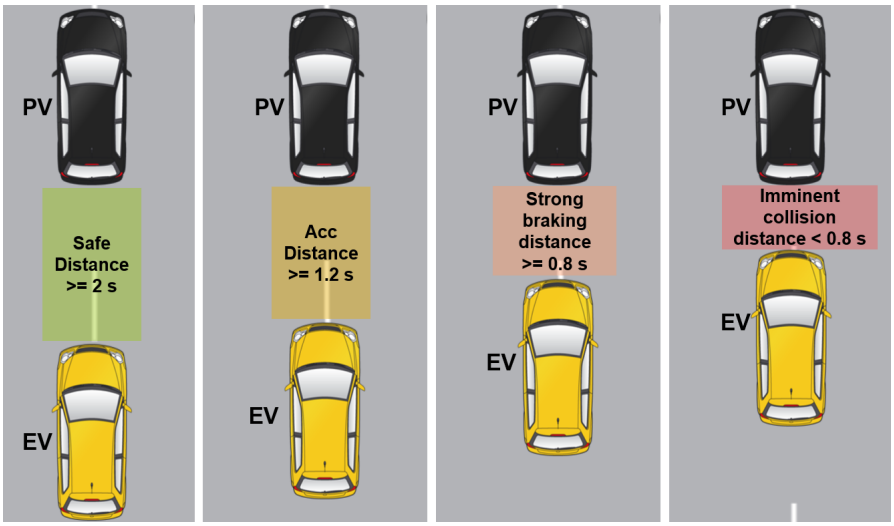

Figure 6.13: Different states of front car distance property measured in seconds.

```
RuleBasedPlanner RBP{
    scene ^Scene
    GOAL goal1
    {
    GoalType Priority
    WHEN{
        GOAL goal2{
            GoalType Constraint
            WHEN{
            GOAL goal3{
            GoalType Constraint
            WHEN{
            WHEN{
                GOAL goal4{
                GoalType Priority
                    WHEN{
                    WHEN{
                    WHEN{
                    WHEN{
                    WHEN{
                }
            }
        }
    }
}
```

Figure 6.14: RuleBasedPlanner file (*Area2Spec.rbp*) containing all the requirements in *Area2Spec*.

Figure 6.14. The **RBP** contains four goals. The **Goal1** has priority goal type (GoalType Priority), which means the condition (WHEN) defined in **Goal1** is executed before all the other conditions (in **Goal1** case, we only have one condition), and before the first condition of the following goal (in this case **Goal2**). If **Goal1** is false, then we pass to **Goal2** that has also one condition (**when**). The **Goal2** has a constraint goal type (GoalType Constraint), which means the condition in **Goal2** is executed in parallel with both conditions defined in **Goal3** since this latter has also a constraint

goal type. **Goal4** has a priority goal type, so we execute the first condition, if it is false we pass to the second one. If the second **when** is false we pass to the third condition. The last condition is executed if the previous one is false.

- **Goal1** detects interference for the front detection of the PV. It also detects interference of the SV and sends back to the engineer an alert of bug notification (seen in Figure 6.15). For the thesis, the action is only informative. Later, actions could be linked with the assistant planning. The RBP check the **Goal1** first that has priority goal type. If there is no false recognition, we execute **Goal2** and **Goal3** in parallel since they have constraint goal type.

```
GOAL goal1
{
GoalType Priority
WHEN{
    NOT(
        OR( //no_interference_in_front_car_detection :
        {   propertyType "EgoProperties.front_car_distance" is
            "EgoProperties.front_car_distance.not_exist"
            ifonlyif(
                OR (propertyType "EgoProperties.front_car_tracking" is
                    "EgoProperties.front_car_tracking.not_exist",
                    propertyType "EgoProperties.front_car_tracking" is
                    "EgoProperties.front_car_tracking.disappeared_less_than_t1",
                    propertyType "EgoProperties.front_car_tracking" is
                    "EgoProperties.front_car_tracking.disappeared_more_than_t1"
                )
            )
        }, //no_interference_in_straddling_car_detection :
        {     propertyType "EgoProperties.straddling_car_distance" is
              "EgoProperties.straddling_car_distance.not_exist"
         ifonlyif (
              propertyType "EgoProperties.straddling_car_tracking" is
              "EgoProperties.straddling_car_tracking.not_exist"
            )
        }
        )
    )
    action "Area2Actions.bug_notification"
    alert "Area2Alerts.bug_notification"
}
```

Figure 6.15: Goal1 in the RBP to detect interferences of PV and SV.

- **Goal2** seen in Figure 6.16 contains one condition to verify disappearance of PV or SV, or even missing line/lane detection when the EV has a stable control. If so, we apply an emergency operation by telling the driver to take over the wheel. The RBP executes at the same time **Goal3**.
- **Goal3** seen in Figure 6.17 contains two conditions executed in parallel. They both report the delta V when EV has an imminent collision distance with PV or SV. We note that **Goal2** and **Goal3** could be sum up in one goal seen that they are executed in parallel.

```
GOAL goal2{
    GoalType Constraint
    WHEN{
        AND(//no lane or infrastructure detection
            propertyType "EgoProperties.stable_control" is
            "EgoProperties.stable_control.stable",
            OR( propertyType "EgoProperties.front_car_tracking" is
            "EgoProperties.front_car_tracking.disappeared_more_than_t1",
                propertyType "EgoProperties.line_detection" is
                "EgoProperties.line_detection.no_detection_in_more_than_t6",
                //swerving SV to EV's lane
                propertyType "EgoProperties.straddling_car_tracking" is
                "EgoProperties.straddling_car_tracking.straddling_more_than_t7"
            )
        )
        action "Area2Actions.emergency_operation_1"
        alert "Area2Alerts.eop1"
    }
```

Figure 6.16: Goal2 in the RBP to check if the system misses detection of PV, SV, or lines while having a stable control and executes an emergency operation.

```
GOAL goal3{
GoalType Constraint //report Delta V when imminent collision
WHEN{
    propertyType "EgoProperties.front_car_distance" is
    "EgoProperties.front_car_distance.imminent_collision_distance"
    action "Area2Actions.report_longitudinal_delta_V"
    alert "Area2Alerts.longitudinal_acceleration"
}
WHEN{
    propertyType "EgoProperties.straddling_car_distance" is
    "EgoProperties.straddling_car_distance.imminent_collision_distance"
    action "Area2Actions.report_longitudinal_delta_V"
    alert "Area2Alerts.longitudinal_acceleration"
}
```

Figure 6.17: Goal3 in the RBP to report the Delta V when imminent collision distance of PV or SV occurs.

– **Goal4** seen in Figure 6.18 starts with testing the imminent collision distance that is considered as the most critical that leads to an emergency maneuver. If a false recognition occurs and during the confirmation of PV, we send the alert to the driver to apply the brake. As mentioned previously, the action could be an immediate control by the rule-based planner. But for now, it is just an alert passed to the driver. If there is no such case, the system tests the rear car distance that imposes the car to not brake in case there is an emergency distance with FV. If not, we verify strong braking and acc distances respectively. Note that the priority is given to the rear car to avoid rear collision, and in case EV faces a critical case of having a front collision, we trigger the emergency maneuver.

```
GOAL goal4{
GoalType Priority
    WHEN{//imminent collision
        OR(propertyType "EgoProperties.front_car_distance" is
           "EgoProperties.front_car_distance.imminent_collision_distance",
           propertyType "EgoProperties.straddling_car_distance" is
           "EgoProperties.straddling_car_distance.imminent_collision_distance")
        action "Area2Actions.emergency_maneuver"
        alert "Area2Alerts.longitudinal_acceleration"
    }
    WHEN{//during confirmation of PV
        propertyType "EgoProperties.front_car_tracking" is
        "EgoProperties.front_car_tracking.not_confirmed"
        action "Area2Actions.brake_acc"
        alert "Area2Alerts.longitudinal_acceleration"
    }
    WHEN{//collision with the rear
        propertyType "EgoProperties.rear_car_distance" is
        "EgoProperties.rear_car_distance.emergency_distance"
        action "Area2Actions.no_action_needed"
        alert "Area2Alerts.no_alert_needed"
    }
    WHEN{//strong braking collision
        OR(propertyType "EgoProperties.front_car_distance" is
           "EgoProperties.front_car_distance.strong_braking_distance",
           propertyType "EgoProperties.straddling_car_distance" is
           "EgoProperties.straddling_car_distance.strong_braking_distance")
        action "Area2Actions.brake_strong"
        alert "Area2Alerts.longitudinal_acceleration"
    }
    WHEN{//acc collision
        OR(propertyType "EgoProperties.front_car_distance" is
           "EgoProperties.front_car_distance.acc_distance",
           propertyType "EgoProperties.straddling_car_distance" is
           "EgoProperties.straddling_car_distance.acc_distance")
        action "Area2Actions.brake_acc"
        alert "Area2Alerts.longitudinal_acceleration"
    }
```

Figure 6.18: Goal4 in the RBP containing five sequential conditions to trigger different type of alerts regarding front and rear car distance violations.

6.5 Area2Spec human-readable generated document using EPSAAV generator

After using the **EPSAAV specifier** to formalize all the environment and rules presented in the previous Section 6.4, the tool manages an automatic generation as discussed in the Chapter 3 in Section 3.3.5.

EPSAAV generator draw up a generated safety report described in Section 3.3.5.1 that can be used as a reference at the corporate level to disseminate the safety policy to be enforced in the company. This document (.txt) is represented in the Appendix A.2. This document is, for now, a text document (.txt) containing the properties, which in our case are the Ego properties, the behaviors which are the action and alert libraries, and the rule-based planner which contains the goals executed in sequence or parallel and following a logical expressiveness. This generation contains same description as *Area2Spec*.
Experts need to know briefly what they have chosen explicitly so as not to confuse others. As mentioned in the workflow using EPSAAV in Figure 6.2, they have

to test if it is necessary to modify the specifications. This is why for the case study where we implemented the *Area2Spec* document, the four written goals are automatically generated in the document (.txt) to summarize the requirements. The users also have the version of each *Area2Spec* environment. After defining their formal requirements, they can simultaneously create and manage multiple versions of the rule-based planner, all of which have the same general safety assessment function but are enhanced, upgraded, or customized. It is also an improvement not only for safety experts but also for developers who need to connect files to the debugger. When a developer is working on the latest files, easy access to goal history helps them understand the purpose of the dataset. It allows him to seamlessly make changes and update the generator that works within the long-term goals of the safety assessment. It tracks contributions made by multiple safety experts and provides traceability as proof of all revisions and changes made. Tracking changes from the original copy to the many improved versions and, finally, to the final version once all the scenarios have been described.

6.6 *Area2Spec* monitor connected to the *FusionRunner*

As mentioned in the Chapter 3 in Section 3.3.5.2 and in Chapter 4, a C monitor is automatically generated, and the files presented in the Section 4.4 are obtained which are then connected to the *FusionRunner* debugger in Section 4.5. We include this monitor for this use case in the Appendix A.3.
In this Section, we present the code in detail and how we connect it to the *FusionRunner* by interfacing and filling all the functions using fusion data.

6.6.1 C code monitor generation using EPSAAV generator

All the generated code is compiled and verified so that the development engineer has only one task to interface with the merge output described in the next Section.

- Area2Actions C code file with its header contains the actions cited in the library in Figure 6.9. Area2Actions.h is the header file presented in Figure 6.19. It contains the enumeration of the actions called **Area2Actions_t**. It also contains a function *processactions()* that only prints out for now the action name.

- Area2Alerts.c and Area2Alerts.h seen in Figure 6.20 contain the alerts in the enumeration **Area2Alerts_t** cited in the library in Figure 6.8. The header file contains a *processalerts()* function that prints out the informative alert to the user.

```
#ifndef Area2Actions_H_
#define Area2Actions_H_

typedef enum {
    ac_brake_acc,
    ac_brake_strong,
    ac_report_longitudinal_delta_V,
    ac_report_lateral_delta_V,
    ac_emergency_operation_1,
    ac_emergency_maneuver,
    ac_bug_notification,
    ac_no_action_needed,
    ac_size,
} Area2Actions_t;

void processactions(Area2Actions_t Area2Actions_enum);

#endif
```

Figure 6.19: Area2Actions.h containing the actions and a *processactions()* function.

```
#ifndef Area2Alerts_H_
#define Area2Alerts_H_

typedef enum {
    al_longitudinal_acceleration,
    al_longitudinal_information_reporter,
    al_lateral_information_reporter,
    al_bug_notification,
    al_eop1,
    al_no_alert_needed,
    al_size,
} Area2Alerts_t;

void processalerts(Area2Alerts_t Area2Alerts_enum);
#endif
```

Figure 6.20: Area2Alerts.h containing the alerts and a *processalerts()* function.

- Safety_Checks.c and Safety_Checks.h are essential to translate formal rules to code using If conditions.
  The first step was generating three functions: (1) *isTriggFunct* boolean function to trigger the goal, (2) *execute* function to call the action, and (3) *raiseAlarm* function to trigger the alert. All of them are used in a **Goal_t**

```
typedef bool_t(isTriggFunct_t)(void);
typedef void(executeBehavior_t)(void);
typedef void(raiseAlarm_t)(void);

typedef struct Goal_t {
    isTriggFunct_t *isTriggered;
    executeBehavior_t *execute;
    raiseAlarm_t *raiseAlarm;
} Goal_t;
```

Figure 6.21: **Goal_t** structure with three functions to trigger, execute, and raise alarms in Safety_Checks.h.

  structure that is then used for each goal with the conditions as a type. We can see an example of the first condition in goal1 in Figure 6.22. The function *isTriggered* is assigned to *trig_goal_condition1()*, *execute* is referred to

```
Goal_t goal1_condition1;
bool_t trig_goal1_condition1();
void  execute_goal1_condition1();
void  alarm_goal1_condition1();
```

Figure 6.22: Declaration of functions for goal1_condition1 in Safety_Checks.h.

execute *goal1_condition1()*, and *raiseAlarm* to *alarm_goal1_condition1()* as seen in Figure 6.23. All these assignments are done in an *init_goals()* function. The

```
goal1_condition1.isTriggered = &trig_goal1_condition1;
goal1_condition1.execute = &execute_goal1_condition1;
goal1_condition1.raiseAlarm = &alarm_goal1_condition1;
```

Figure 6.23: Example of an instantiation of goal1_condition1 in *init_goals()* function in Safety_Checks.c.

trig_goal_condition1() function contains the if condition of all the properties written in the RBP. Figure 6.24 shows the transformation given by the monitor. To check states for the properties, we create for each property a check

```
bool_t trig_goal1_condition1() {
    if ( !( (front_car_distance_check() == not_exist_front_car_distance &&
            (front_car_tracking_check() == not_exist_front_car_tracking ||
            (front_car_tracking_check() == disappeared_less_than_t1_front_car_tracking )
            || (front_car_tracking_check() == disappeared_more_than_t1_front_car_tracking )))
            || (straddling_car_distance_check() == not_exist_straddling_car_distance &&
                  (straddling_car_tracking_check() == not_exist_straddling_car_tracking )))) {
        goal1_condition1.execute();
        goal1_condition1.raiseAlarm();
        return TRUE;
    }
    return FALSE;
}
```

Figure 6.24: *trig_goal1_condition1()* function using if condition to check safety in Safety_Checks.c.

function and define them in EgoProperties C files. *Trig_goal1_condition1()* calls the actions and the alerts if the conditions are true. If not, the function returns false. The other goals and conditions are shown in the Appendix. An important notion given by the monitor is the type of execution that is also generated depending on the goal type that the safety engineer inserted (Constraint or Priority). Figure 6.25 shows the *trig_goals()* defining the priorities between all goals. Goal1_condition1 has a priority over all other goals. **Goal2** and **goal3** are executed in parallel. **Goal4** has a sequential execution between its conditions.

- EgoProperties.h lists states in enumeration types. Figure 6.26 is an example for the **traffic_jam** and **front_car_distance** properties enumerating all possible states defined in the library by the safety engineer. It

```
void trig_goals() {
    if (goal1_condition1.isTriggered() == FALSE ) {
        goal2_condition1.isTriggered();
        goal3_condition1.isTriggered();
        goal3_condition2.isTriggered();
        if ( goal4_condition1.isTriggered() == FALSE ) {
            if ( goal4_condition2.isTriggered() == FALSE ) {
                if ( goal4_condition3.isTriggered() == FALSE ) {
                    if ( goal4_condition4.isTriggered() == FALSE ) {
                        goal4_condition5.isTriggered();
                    }
                }
            }
        }
    }
}
```

Figure 6.25: *trig_goals()* defining the priorities and parallel executions between all goals in Safety_Checks.c.

also declares functions for each property such as *traffic_jam_check()* and *front_car_distance_check()*.

```
#include <math.h>
#ifndef EgoProperties_H_
#define EgoProperties_H_

typedef enum {
    yes_traffic_jam,
    no_traffic_jam,
} traffic_jam_t;

typedef enum {
    not_exist_front_car_distance,
    safe_distance_front_car_distance,
    acc_distance_front_car_distance,
    strong_braking_distance_front_car_distance,
    imminent_collision_distance_front_car_distance,
} front_car_distance_t;
```

Figure 6.26: Enumerations of states for each property in EgoProperties.h.

EgoProperties.c details functions that the developer must fill to bridge the check functions with the current state. Figure 6.27 shows what is generated by the monitor. The commented lines in green are the parts that must be connected to the debugger by the developer and given back to the safety engineer.

- Scene.h contains object parameters for each object type (Figure 6.28). It also intantiates the getter functions for each parameter to each object type.

- Scene.c contains the roles defined using object types such as PV and SV that used **Obstacle_t** data type(Figure 6.29). *FusionDatatoScene()* function is created to bridge the ouptut of the monitor generated with the input of fusion data in the debugger. The *getParameters()* functions for each role object are to be filled by the developer.

```
front_car_distance_t front_car_distance_check() {
    /*TODO User Adaptation : functions declared in Scene.c with thresholds
    static const float safe_distance_threshold = 2.0f; //safe_distance>=2.0s
    static const float acc_distance_threshold = 1.2f; //acc_distance>=1.2s
    static const float strong_braking_distance_threshold = 0.8f; //strong_braking_distance>=0.8s
    if () {
        return not_exist_front_car_distance;
    }else if () {
        return safe_distance_front_car_distance;
    }else if () {
        return acc_distance_front_car_distance;
    }else if () {
        return strong_braking_distance_front_car_distance;
    } else {
        return imminent_collision_distance_front_car_distance;
    }
    */
    return not_exist_front_car_distance;
}
```

Figure 6.27: Example of *front_car_distance_check()* function to check for the state in EgoProperties.c.

```
typedef struct Ego {
    float_t Lane_detection;
    float_t Lane_missing;
} Ego_t;

typedef struct Obstacle {
    uint32_t id;
    float_t counter_tracking;
    float_t counter_not_tracking;
    float_t ttc_min;
} Obstacle_t;
```

Figure 6.28: **Ego_t** and **Obstacle_t** object types enumerating parameters in Scene.h.

```
static Ego_t Ego;
static Obstacle_t preceding_vehicle;
static Obstacle_t straddling_vehicle;

static void Ego_getParameters(/*TODO User Adaptation : Parameter FusionControlData */);
static void preceding_vehicle_getParameters(/*TODO User Adaptation : Parameter FusionControlData */);
static void straddling_vehicle_getParameters(/*TODO User Adaptation : Parameter FusionControlData */);

void fusionDatatoScene(/*TODO User Adaptation : Parameter FusionControlData */) {
    Ego_getParameters(/*TODO User Adaptation : Parameter FusionControlData */);
    preceding_vehicle_getParameters(/*TODO User Adaptation : Parameter FusionControlData */);
    straddling_vehicle_getParameters(/*TODO User Adaptation : Parameter FusionControlData */);
}
```

Figure 6.29: Perceived objects instantiation in the scene and function to link the output of the monitor to the input of the debugger in Scene.c.

6.6.2 *Area2Spec Safety Checker module* connected to the *FusionRunner*

The two main files that need to be adapted concern applying the thresholds for all states in the properties (in EgoProperties.c), and defining properties check functions by getting the corresponding parameters from the *FusionRunner* (in Scene.c).

The values and thresholds are chosen by the safety expert according to the specifications of the given AEB braking system. In EgoProperties.c, we can cite all check functions for all the properties:

- We start with the *traffic_jam_check()* function. Since the ODD requires the EV to always be operational in a traffic jam, we only return a true value for this function.
- For *front_car_distance_check()* function, we use *get_preceding_vehicle_ttc_min()* to get the TTC value in seconds and compare it with the states. If it is 2 seconds or more, it is a **safe_distance**, else if it is more or equal to 1.2 seconds it is an **acc_distance**, else if TTC is more or equal to 0.8 seconds, it is strong braking, else it is an **imminent_collision_distance**.
- For *front_car_tracking_check()* function, we consider the threshold t1=0.25 seconds when the front car disappeared. We created two functions: one to track the existence of PV named *get_preceding_vehicle_counter_tracking()*, and one *get_preceding_vehicle_counter_not_tracking()* whenever PV is missing.
- For *straddling_car_distance_check()* function, same thresholds are used as for the function *front_car_distance_check()* using *get_straddling_vehicle_ttc_min()* function.
- For *straddling_car_tracking_check()* function, same as *front_car_tracking_check()* with a t7= 2 seconds threshold. The function that we generate to track the existence of SV is *get_straddling_vehicle_counter_tracking()*.
- For *line_detection_check()* function, the threshold to consider the line disappeared is t6= 0.3 seconds. We created two functions: one to track the existence of the lanes named *get_Ego_Lane_detection()*, and another function if lanes are missing *get_Ego_Lane_missing()*, since FusionRunner simplify the tasks by giving directly the option to use lane informartion.
- For *rear_car_distance_check()* function, we return **not_exist_rear_car_distance** since scenarios tested in the *FusionRunner* do not consider FVs.
- For *stable_control_check()* function, we consider EV always stable returning true state since the Fusion is already done of the set of data and information is filtered to have better stability in the scenarios.

In Scene.c, we all have getter functions for information connected by Fusion. The Figure 6.30 represents the positions of all the obstacles which can be SV or PV. The SV can be Obstacle2, Obstacle3, Obstacle6 or Obstacle7. PVs can be Obstacle0 or Obstacle1. Many important information for security analysis such as TTC are provided by the Fusion.

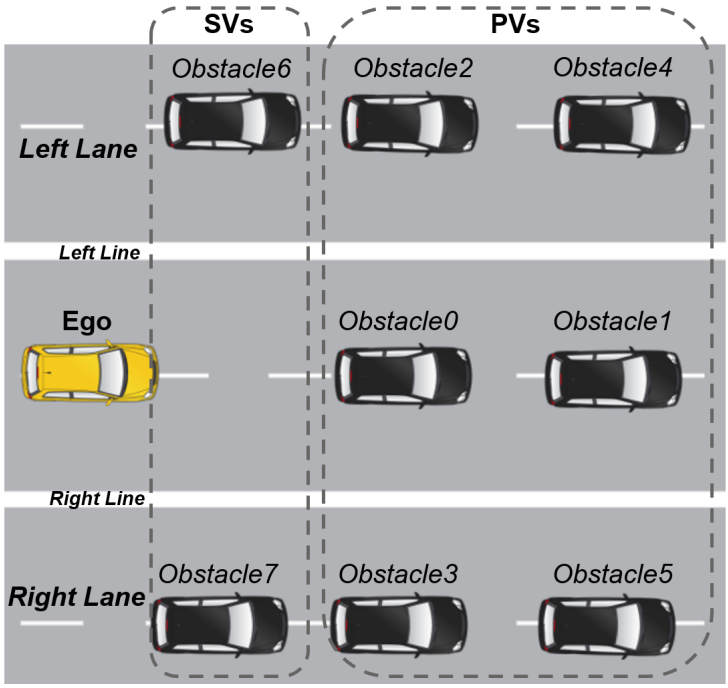

Figure 6.30: Obstacle and Ego positions in the scene defined in the *FusionRunner*.

- For the *Ego_getParameters()*, we need to check lane detection that is composed of right and left lines. If interferences occur, then we affect **Lane_detection** and **Lane_missing** parameters.

- For the *preceding_vehicle_getParameters()*, if we do not detect any Obstacle0 position, we increment the **counter_not_tracking** parameter. If the scenario starts tracking the PV with an id corresponding to the Obstacle0 position, then we take the corresponding ID and TTC values, and we start affecting the **counter_tracking** parameter.

- For the *straddling_vehicle_getParameters()*, if we do not detect any position regarding Obstacle2, Obstacle3, Obstacle6, or Obstacle7, we increment the **counter_not_tracking** parameter, and if the scenario starts tracking an SV with an id corresponding to the cited obstacles, then we take the id and ttc values, and we start affecting the **counter_tracking** parameter. If there exist multiple SVs, one SV is chosen with the least value of TTC.

6.6.3 Testing scenarios on real data recordings

After generating the C monitor and connecting it with FusionRunner, we replay real-data drives that are recorded and start analyzing them. In this Section, we detail some analysis and describe the source of violations.

6.6.3.1 Scenario 1: no lane detection

A scenario is given in 6.31 where the ego car is not detecting the lines even when there is stability in the control of the vehicle, and this is due to bad markings on the ground. By stability, we know that there are no lines and the error does not come from the sensors.

The Fusion Display in Figure 6.32 shows the sketch in a window with the

Figure 6.31: No lane/line detection in the Fusion Context View.

corresponding parameters for this step, in which we see that the car with ID = 25766, is entering ego's lane.

At step=2594, this car is not perceiving any line (see Figure 6.33). It is good to note that fusion algorithm gives us information about lanes, which makes the procedure easier to track the lines.

The safety assessment for this scenario is described in **goal2_cond1** to test the missing lane detection (see Figure 6.16) that leads to trigger an action and an alert of **EOP1** (Emergency Operation) as seen in Figure 6.34.

6.6.3.2 Scenario 2: strong braking collision with PV

A scenario is given in 6.35 where the ego car is having a strong braking distance with a PV.

The Fusion Display in Figure 6.36 shows the sketch in a window with the corresponding parameters for this step, in which we see that the PV's distance is a strong braking distance.

At step=540, the EV has a TTC=0.815, which is a previously defined threshold for the PV distance described in Figure 6.13(see Figure 6.37). It is good to note that the fusion algorithm gives us information about the lanes, which helps in the procedure of tracking the lines.

The safety assessment for this scenario is described in **goal4_cond4** to test the strong braking distance with the PV (see Figure 6.18) that leads to trigger an action and an alert of **Brake Strong** and **Longitudinal Acceleration** respectively as seen in Figure 6.38.

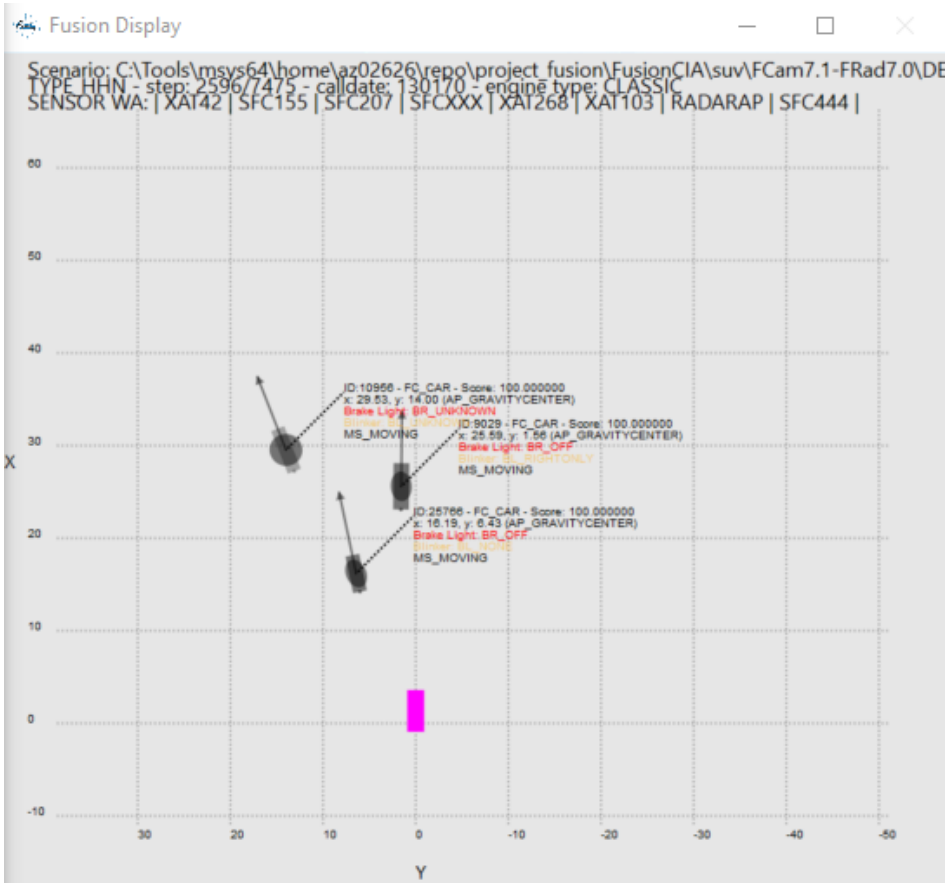

Figure 6.32: Fusion Display window showing the missing lanes.

6.6.3.3 Scenario 3: No rule violations

A scenario is given in 6.39 where the ego car is not violating any safety rule.

The Fusion Display in Figure 6.40 shows the sketch in a window with the corresponding parameters for this step, in which we see that the ego car is perceiving a front vehicle.

At step=6617, this car is in a safe situation having a stable front car tracking and a line stability (see Figure 6.41).

6.7 *Area2Spec* verification engine fed to the SAT solver

EPSAAV generator gives a verification engine that produces three Java files as mentioned in Chapter 3 in Section 3.3.5.3, and detailed in Chapter 5:

- Sat4jRules.java that contains goals expressed in boolean specification func-

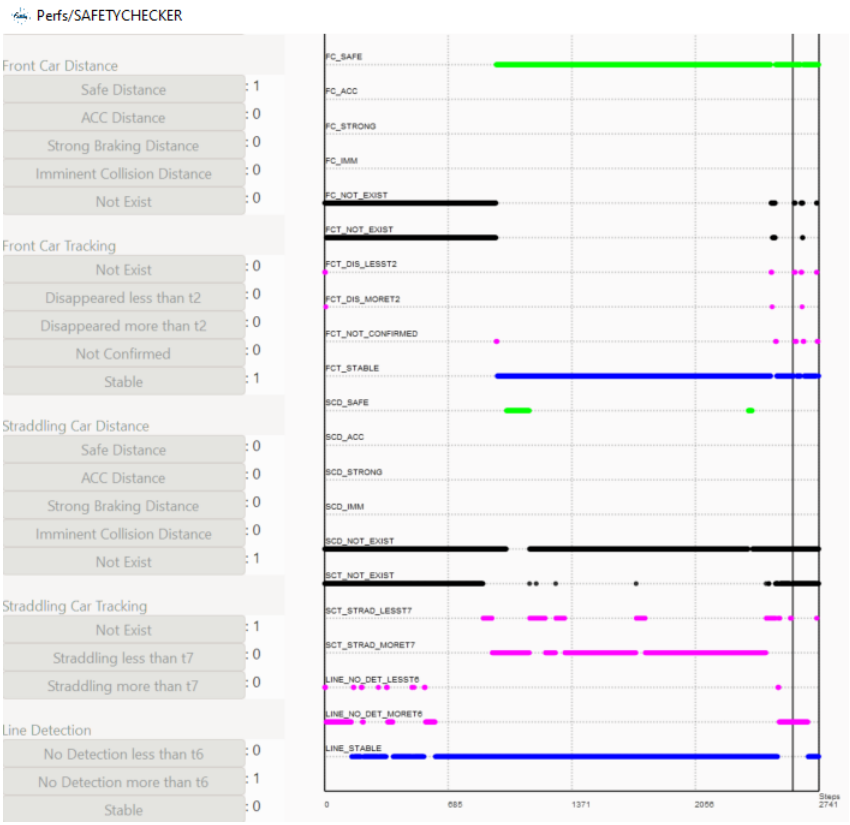

Figure 6.33: SAFETYCHECKER window showing states triggered for properties for step=2594.

tions,

- Sat4jRulesConsistency.java that forces the generated functions in the previous file to be true and false, and shows the generated code to test priority or parallel executions,

- and Sat4jSystemConsistency.java that tests all solutions for the rule-based planner with the coherence of goals and properties.

We include the generated Java files in the Appendix A.4. Since we have in *Area2Spec* four goals containing 9 conditions, we get 9 build functions that gave us a translation for the logical requirements. Figure 6.42 is an example of *build_goal1_cond1()* and *build_goal2_cond1()* functions in Sat4jRules.java. We have 45 solutions for the **goal1_cond1**; 15 solutions for **goal2_cond1**; one solution for each **goal3_cond1**, **goal3_cond2**, **goal4_cond2**, and **goal4_cond3**; and three solutions for each **goal4_cond1**, **goal4_cond4**, and **goal4_cond5**.

In Sat4jRulesConsistency.java, we generate two functions for each goal as seen in Figure 6.43. *Build_goal1_cond1_True()* is created to force the **goal1_cond1** to

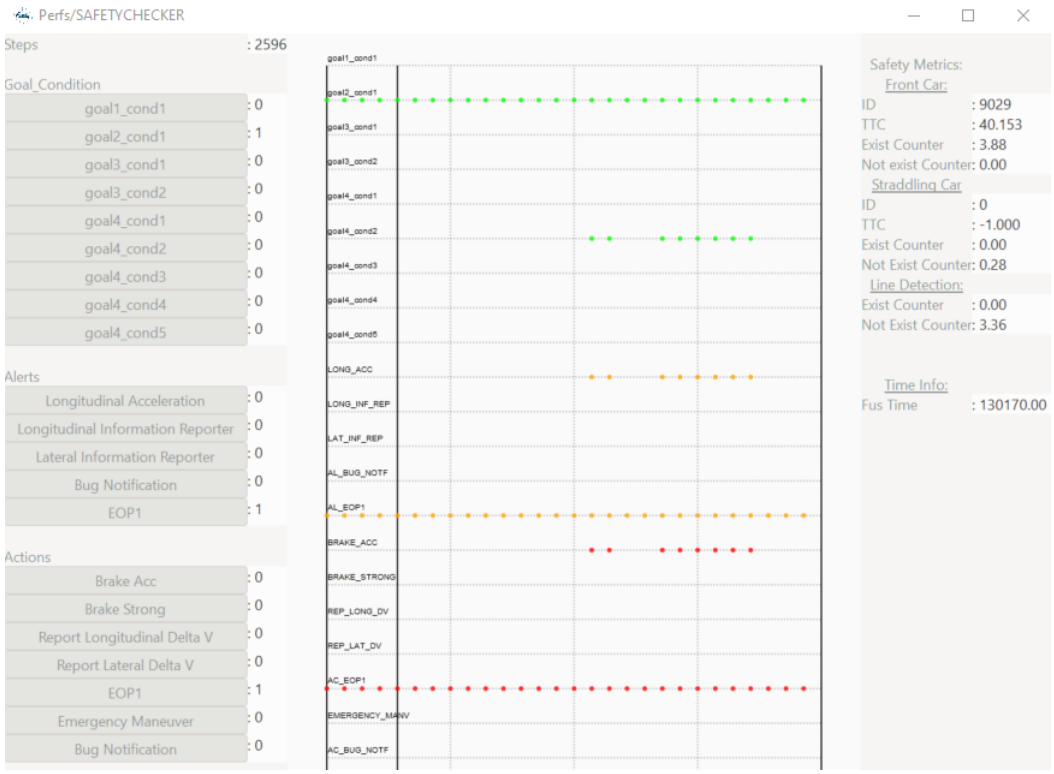

Figure 6.34: SAFETYCHECKER window showing **goal2_cond1** and corresponding behaviors triggered for step=2594.

Figure 6.35: Strong braking distance with a PV in the Fusion Context View.

be true, and *build_goal1_cond1_False()* to be false. We apply the same procedure for all conditions in each goal. **Goal2_cond1** presents 7 true solutions out of 15, that test the system if EV is performing in a stable control, and there test the disappearance of SV, PV, or even in line detection to perform an emergency maneuver.

In *Area2Spec* example, *build_goal1_cond1_True()* has 29 results and *build_goal1_cond1_False()* has 16 which makes them 45 in total. We also generate goals with triggered behaviors so that we can study behaviors inconsis-

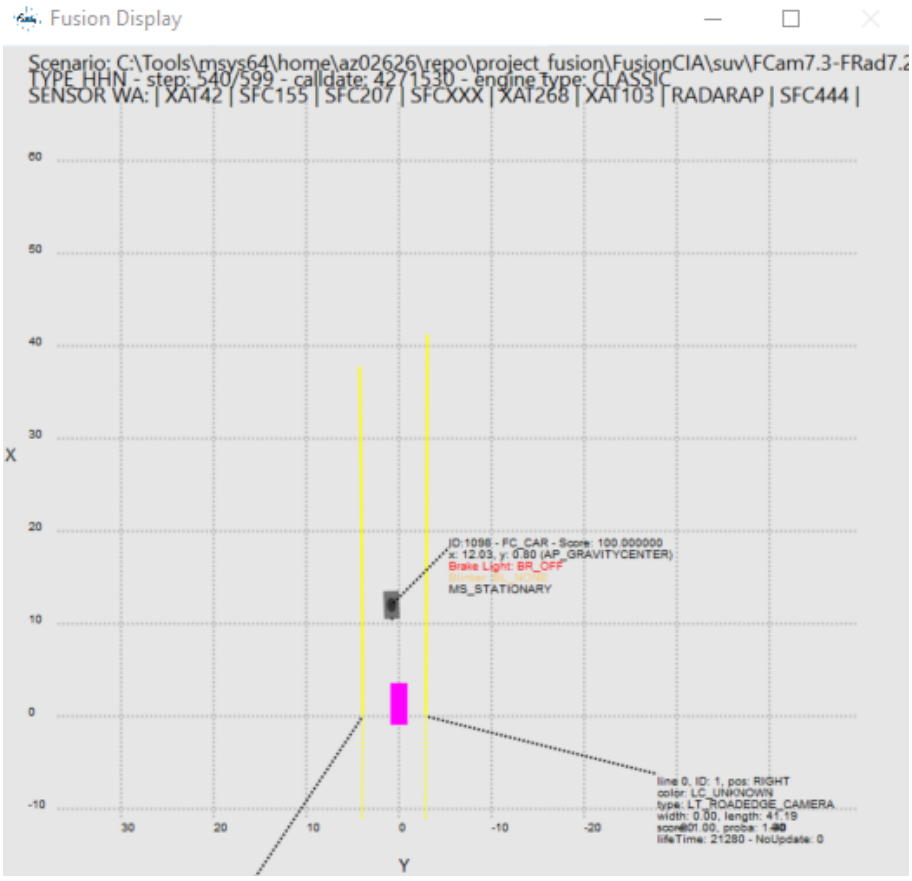

Figure 6.36: Fusion Display window showing the missing lanes.

tency. In Figure 6.44, we show the three functions for alerts generation. The first *build_goal1_cond1_Alert()* (90 solutions=29 true + 61 false) which takes the *build_goal1_cond1()* from Sat4jRules.java file. *True* and *False* functions are also generated for the alerts.
For the priority and parallel tests, we apply ($\neg$X$\wedge$Y) and (X$\vee$Y) tests respectively. **Goal1_cond1** has a priority execution over **Goal2_cond1**. Since **Goal1_cond1** starts with a NEGATION operator, the negation of it is true. Figure 6.45 shows the priority test code trying $\neg$**Goal2_cond1** $\vee$ **Goal1_cond1**.
Goal2_cond1 has a parallel execution with **Goal3_cond1**. Figure 6.46 shows the parallel test code trying **Goal2_cond1** $\wedge$ **Goal3_cond1**. Both of the goals should be set to true and executed at the same time.

In Sat4jSystemConsistency.java, we generate functions to test the system solutions (seen in Figure 6.47). The system solution is the multiplication of the coherence of the properties and of all goals. By multiplication, we mean the AND operator. Since we have four goals, we generate a function for each goal called *coherence_goal()* that takes the condition to help execute the goal. For

Figure 6.37: SAFETYCHECKER window showing states triggered for properties for step=2594.

instance, **goal2_cond1** is true when **goal1_cond1** is true (since **goal1_cond1** is originally false). So the coherence of **goal2_cond1** is achieved when we apply the AND operator between this goal and the *goal1_cond1_True()*.

The coherence of the properties is the addition of all coherence of each property as seen in Figure 6.48. For instance, the coherence of **traffic_jam** is when one of the states is triggered, such as we can not have both states (no and yes) for a traffic jam at the same time. Same procedure is applied for all the other properties. This is how we can manage to check on solutions for the system and analyze what cases we can have and what we can edit. For *Area2Spec*, we can have 7128 solutions for the system respecting properties and goal coherences, which 3600 of them are true.

6.8 SAVI process

The Safety Analysis of Violations and Inconsistencies (SAVI) process involves modifying requirements after detecting ambiguities or inconsistencies in the rules. Ambiguities are investigated using *FusionRunner* to visualize real or simulated

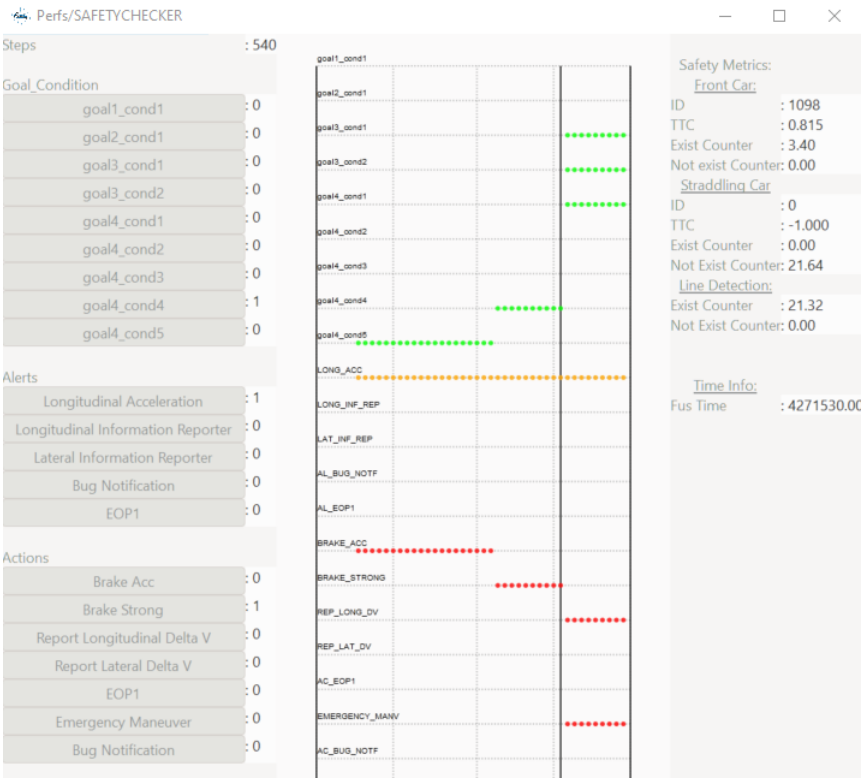

Figure 6.38: SAFETYCHECKER window showing goals and behaviors triggered for step=2594.

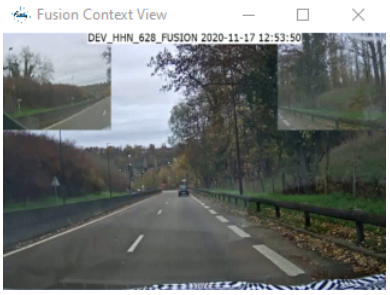

Figure 6.39: No rule violations in the Fusion Context View.

data, and inconsistencies are tested using the SAT4J solver. The SAVI process follows the *Area2Spec* document, where we generate from the **EPSAAV generator** a C monitor and a Java verification engine. These two are the inputs to the process, and the outputs are tested again to assess safety.

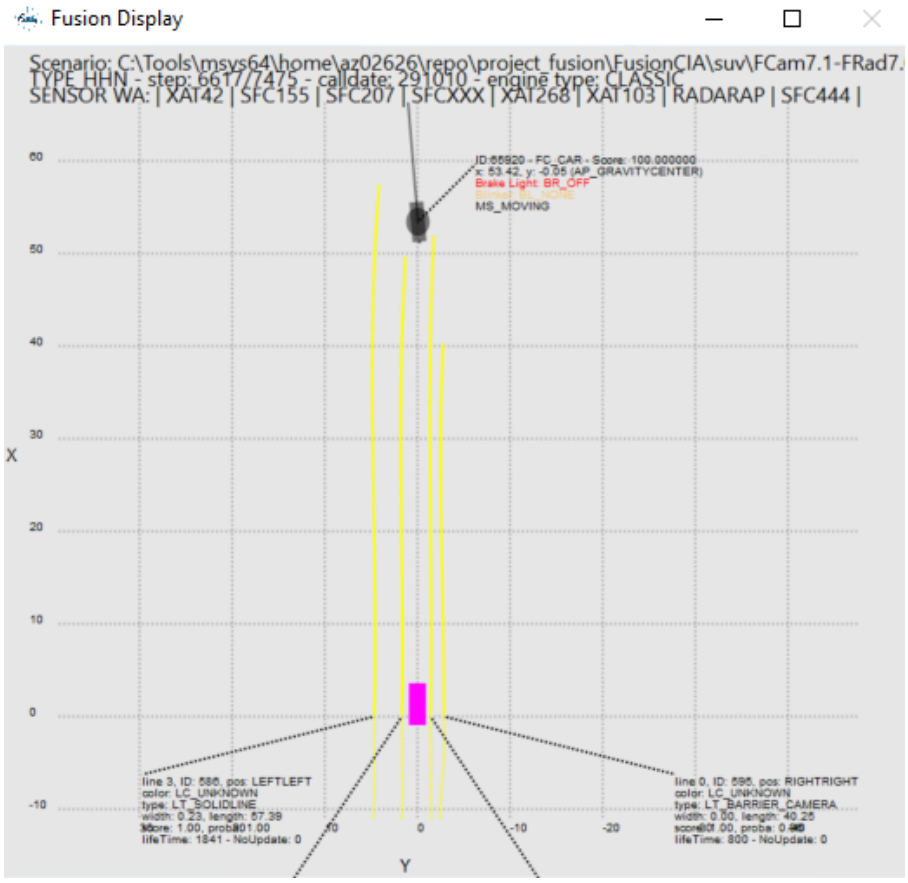

Figure 6.40: Fusion Display window showing the tracking of the PV with no violation.

6.8.1 Ambiguities detection

We test Safety Module generated from *Area2Spec* requirements on real data scenarios using the *FusionRunner*. Among these scenarios, we have open road driving data, which includes city, and highway environments. In addition, we also have NCAP real data recordings that a predetermined real use cases [31].

The advantage of using *FusionRunner* is that we can track violations and state values graphically. One of these violations that caught the eyes is in the data set at step=4217 and refers to the **goal2_cond1** as seen in 6.49. The behaviors triggered are **emergency_operation**. The states that are true are: **not_exist_front_car_distance**, **disappeared_more_than_t2_front_car_tracking**, **not_exist_straddling_car_distance**, **not_exist_straddling_car_tracking**, and **stable_lne_detection**.
If we look at Figure 6.50, we do not see PV or SV in the real video.
This is considered a detected ambiguity in the wrong formalization of the rule, and if we go back to the formal requirement of Figure 6.16, the goal should be

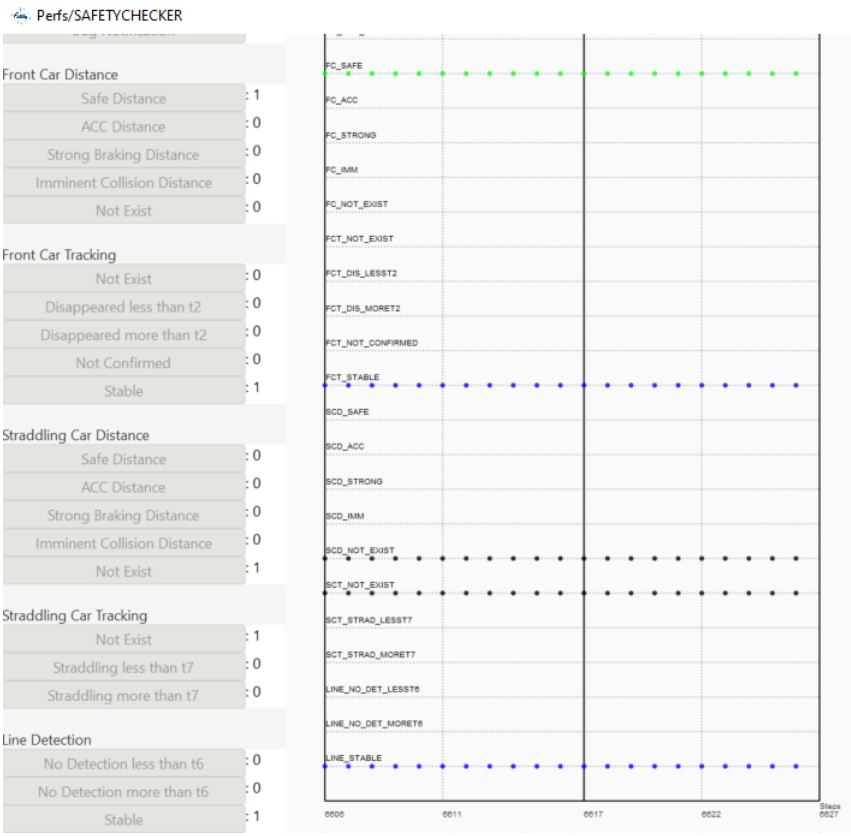

Figure 6.41: SAFETYCHECKER window showing states triggered for properties.

```
static public void build_goal1_cond1(IBooleanSpecification spec) {
    //spec.not("goal1_cond1");
    spec.or("goal1_cond1", "goal1_cond1_1", "goal1_cond1_2");
    build_Equiv(spec,"goal1_cond1_1", "not_exist_front_car_distance",
                "goal1_cond1_1_1");
    spec.or("goal1_cond1_1_1", "not_exist_front_car_tracking",
            "disappeared_less_than_t1_front_car_tracking",
            "disappeared_more_than_t1_front_car_tracking");
    build_Equiv(spec,"goal1_cond1_2", "not_exist_straddling_car_distance",
                "not_exist_straddling_car_tracking");
}

static public void build_goal2_cond1(IBooleanSpecification spec) {
    spec.and("goal2_cond1", "stable_stable_control", "goal2_cond1_1");
    spec.or("goal2_cond1_1",  "disappeared_more_than_t1_front_car_tracking",
            "no_detection_in_more_than_t6_line_detection",
            "straddling_more_than_t7_straddling_car_tracking");
}
```

Figure 6.42: *build_goal1_cond1()* and *build_goal2_cond1()* functions for **goal1_cond1** and **goal2_cond1** representing boolean translation for logical operations in Sat4jRules.java.

```
static public void build_goal1_cond1_True(IBooleanSpecification spec) {
    Sat4jRules.build_goal1_cond1(spec);
    spec.forces("goal1_cond1");
}

static public void build_goal1_cond1_False(IBooleanSpecification spec) {
    Sat4jRules.build_goal1_cond1(spec);
    spec.not("goal1_cond1");
}
```

Figure 6.43: *build_goal1_cond1_True()* and *build_goal1_cond1_False()* functions for **goal1_cond1** in Sat4jRulesConsistency.java.

```
static public void build_goal1_cond1_Alert(IBooleanSpecification spec) {
    Sat4jRules.build_goal1_cond1(spec);
    spec.and("goal1_cond1_Alert", "goal1_cond1","bug_notification");
}

static public void build_goal1_cond1_Alert_True(IBooleanSpecification spec) {
    build_goal1_cond1_Alert(spec);
    spec.forces("goal1_cond1_Alert");
}

static public void build_goal1_cond1_Alert_False(IBooleanSpecification spec) {
    build_goal1_cond1_Alert(spec);
    spec.not("goal1_cond1_Alert");
}
```

Figure 6.44: Alert functions for **goal1_cond1** in Sat4jRulesConsistency.java.

edited by adding a condition to execute the emergency maneuver. The modification is seen in Figure 6.51 which consists of adding a condition on the line stability that avoids triggering error for such case.

6.8.2 Rule inconsistencies verification

In this part, we analyze all solutions for the generated verification engine using the SAT4J library. For the solutions of **goal1_cond1**, 16 solutions out of 45 are false which makes this rule true since it has the NEGATION operator. These solutions constitute the possible cases where we can have a bug. Figure 6.52 shows a solution where **front_car_tracking** (variable 7) and **strad-**

```
static public void testPrioritygoal2_cond1_over_goal1_cond1(IBooleanSpecification spec) {
    build_goal1_cond1_True(spec);
    build_goal2_cond1_True(spec);
}

static public void testPrioritygoal2_cond1_over_goal1_cond1_Alerts(IBooleanSpecification spec) {
    build_goal1_cond1_Alert_True(spec);
    build_goal2_cond1_Alert_True(spec);
}
```

Figure 6.45: Priority test for **Goal2_cond1** and **Goal1_cond1** considering or not alerts in Sat4jRulesConsistency.java.

```
static public void testParallelgoal3_cond1_goal3_cond2(IBooleanSpecification spec) {
    build_goal3_cond2_True(spec);
    build_goal3_cond1_True(spec);
}

static public void testParallelgoal3_cond1_goal3_cond2_Alerts(IBooleanSpecification spec) {
    build_goal3_cond2_Alert_True(spec);
    build_goal3_cond1_Alert_True(spec);
}
```

Figure 6.46: Parallel tests for **Goal2_cond1** and **Goal3_cond1** considering or not alerts in Sat4jRulesConsistency.java.

```
    static public void system_solution(IBooleanSpecification spec){
        coherence_all_properties(spec);
        coherence_all_goals_behaviors(spec);
        spec.and("system_solution", "coherence_all_properties", "coherence_all_goals_behaviors");
    }

    static public void coherence_all_goals_behaviors(IBooleanSpecification spec){
        coherence_goal1(spec);
        coherence_goal2(spec);
        coherence_goal3(spec);
        coherence_goal4(spec);
        spec.or("coherence_all_goals_behaviors", "coherence_goal1","coherence_goal2",
                "coherence_goal3","coherence_goal4");
    }

    static public void coherence_all_goals_behaviors_true(IBooleanSpecification spec){
        coherence_all_goals_behaviors(spec);
        spec.forces("coherence_all_goals_behaviors");
    }
```

Figure 6.47: System solution that is the coherence of all four goals and coherence of all properties in Sat4jSystemConsistency.java.

dling_car_tracking (variable 11) are not compatible with **front_car_distance** (variable 4) and **straddling_car_distance** (variable 10) respectively, which cause triggering a bug notification.

After analyzing the 16 solutions, we noticed that this goal only presents verification of both **front_car** and **straddling_car** incompatibilities and not just one of them. This can be fixed by proposing a replacement of the OR operator in **goal1_cond1**(seen in Figure 6.15) with the AND operator that includes the conflict in each obstacle type.

Figure 6.53 shows also a bug having **disappeared_more_than_t1** and **disappeared_less_than_t1** (variables 8 and 9) at the same time. We can also eliminate this issue by testing **goal1_cond1** with the coherence of **front_car_tracking** property (Figure 6.54a) that helps us deleting unreasonable cases (Figure 6.54b).

The first function in Figure 6.45 presents 103 solutions that show when both sequential rules are applied at the same time. To eliminate having more than one state set to true for the properties used, we apply *coherence_front_car_tracking()* and *coherence_straddling_car_tracking()* which give us 29 solutions. This means

```
static public void coherence_all_properties(IBooleanSpecification spec){
    coherence_traffic_jam(spec);
    coherence_front_car_distance(spec);
    coherence_front_car_tracking(spec);
    coherence_line_detection(spec);
    coherence_straddling_car_distance(spec);
    coherence_straddling_car_tracking(spec);
    coherence_stable_control(spec);
    coherence_rear_car_distance(spec);
    spec.or("coherence_all_properties", "coherence_traffic_jam",
            "coherence_front_car_distance", "coherence_front_car_tracking",
            "coherence_line_detection", "coherence_straddling_car_distance",
            "coherence_straddling_car_tracking", "coherence_stable_control");
}

static public void coherence_traffic_jam(IBooleanSpecification spec){
    spec.clause("yes_traffic_jam", "no_traffic_jam","coherence_traffic_jam");
    spec.clause("no_traffic_jam", "yes_traffic_jam","coherence_traffic_jam");
    spec.clause("coherence_traffic_jam", "yes_traffic_jam", "no_traffic_jam");
    spec.forbids("yes_traffic_jam","no_traffic_jam");
    spec.forbids("no_traffic_jam","yes_traffic_jam");
    spec.not("coherence_traffic_jam");
}
```

Figure 6.48: *Coherence_all_properties()* and *coherence_traffic_jam()* functions in Sat4jSystemConsistency.java.

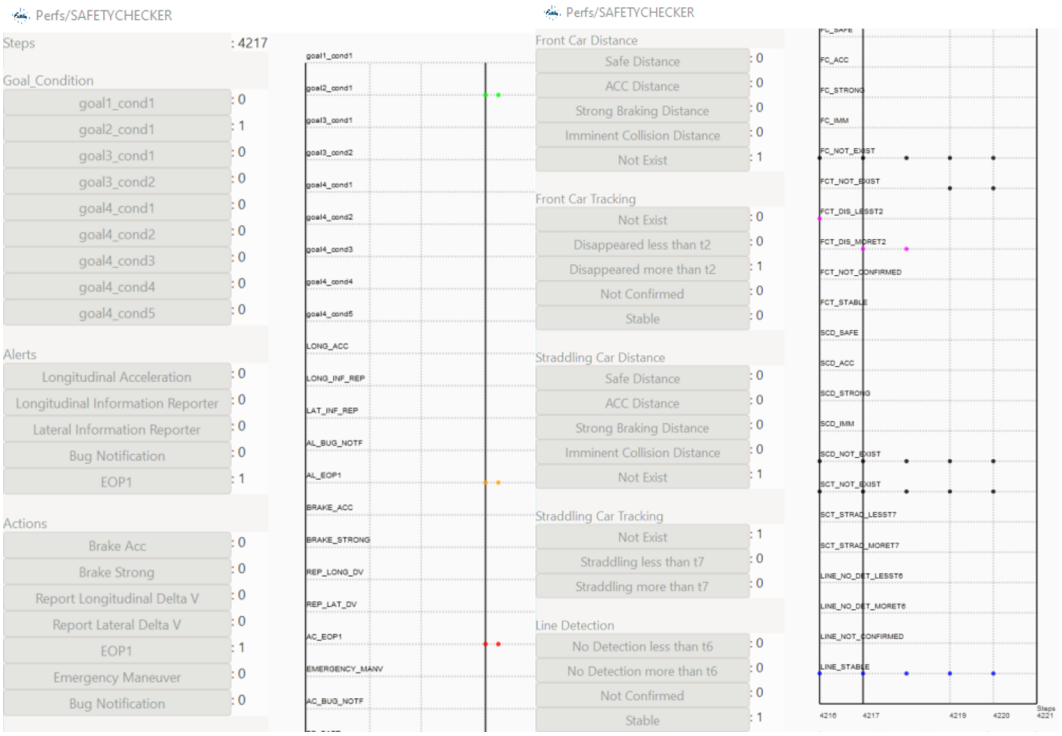

Figure 6.49: Emergency maneuver triggered in a real replayed scenario.

that when **goal1_cond1** is not true (we use the *True* function since this goal starts with a NOT operator), **goal2_cond1** can be true. The engineer can now

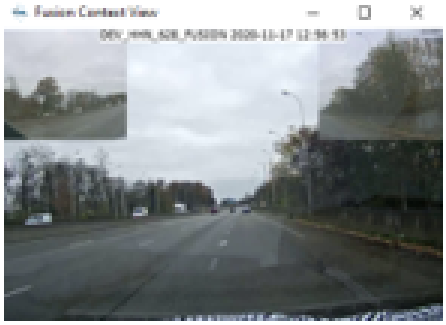

Figure 6.50: Ambiguity in a real replayed scenario at step=4217 triggering emergency maneuver with no PV.

```
GOAL goal2{
    GoalType Constraint
    WHEN{
        AND(//no lane or infrastructure detection
            propertyType "EgoProperties.stable_control" is
            "EgoProperties.stable_control.stable",
            AND(
                NOT(propertyType "EgoProperties.line_detection" is "EgoProperties.line_detection.stable"),
                OR( propertyType "EgoProperties.front_car_tracking" is
                "EgoProperties.front_car_tracking.disappeared_more_than_t1",
                    propertyType "EgoProperties.line_detection" is
                    "EgoProperties.line_detection.no_detection_in_more_than_t6",
                    //swerving SV to EV's lane
                    propertyType "EgoProperties.straddling_car_tracking" is
                    "EgoProperties.straddling_car_tracking.straddling_more_than_t7"
                )
            )
        )
        action "Area2Actions.emergency_operation_1"
        alert "Area2Alerts.eop1"
    }
```

Figure 6.51: Ambiguity solved in the real replayed scenario at step=4217 adding a non stability line condition.

```
Solution 1:
[-1, -2, -3, 4, -5, 6, -7, -8, -9, 10, -11, 12]
1: goal1_cond1=0,
2: goal1_cond1_1=0,
3: goal1_cond1_2=0,
4: not_exist_front_car_distance=1,
5: goal1_cond1_1_1=0,
6: not_exist_front_car_distancegoal1_cond1_1_1=1,
7: not_exist_front_car_tracking=0,
8: disappeared_less_than_t1_front_car_tracking=0,
9: disappeared_more_than_t1_front_car_tracking=0,
10: not_exist_straddling_car_distance=1,
11: not_exist_straddling_car_tracking=0,
12: not_exist_straddling_car_distancenot_exist_straddling_car_tracking=1,
```

Figure 6.52: Solution for **goal1_cond1** presenting an inconsistency in distance and tracking variables.

pass by the solutions to see if there is a case where the second rule should be executed before the first one. If so, current requirements should be edited. Same priority test is applied for **goal4_cond1** and **goal4_cond2** (seen in Figure 6.18).

```
Solution 15:
[-1, -2, -3, -4, 5, 6, -7, 8, 9, -10, 11, 12]
1: goal1_cond1=0,
2: goal1_cond1_1=0,
3: goal1_cond1_2=0,
4: not_exist_front_car_distance=0,
5: goal1_cond1_1_1=1,
6: not_exist_front_car_distancegoal1_cond1_1_1=1,
7: not_exist_front_car_tracking=0,
8: disappeared_less_than_t1_front_car_tracking=1,
9: disappeared_more_than_t1_front_car_tracking=1,
10: not_exist_straddling_car_distance=0,
11: not_exist_straddling_car_tracking=1,
12: not_exist_straddling_car_distancenot_exist_straddling_car_tracking=1,
```

Figure 6.53: Solution for **goal1_cond1** presenting an inconsistency in the states of the same property.

We notice one solution (Figure 6.55) where the **not_confirmed** state is set to true, and the **imminent_collision_distance** is null. The safety engineer can analyze this solution by asking if the **not_confirmed** case can include or not the imminent collision case. This problem is discussed in Chapter 5 in equation 5.7. If so, a modification is required since variable 4 is included in variable 2.

Figure 6.56 represents one solution out of 7 when parallel test is done on **goal2_cond1** and **goal3_cond1**. This solution contains an inconsistency regarding PV properties; we can see **disappeared_more_than_t1_front_car_tracking** false (variable 4) while **imminent_collision_distance_front_car_distance** is true (variable 8). The safety expert shall reconsider these requirements that can affect the system's decision.

The consistency of each goal with each other is tested in Sat4jSystemConsistency.java, as are the system-wide solutions where we can find all the possibilities. The coherence of the whole system is equal to the coherence of all the goals. By carrying out all the previous tests on the rules, the sequential rules, and the parallel rules, it is ensured that the system is coherent and that the rules do not contain any inconsistencies.

6.9 Conclusion

The safety and security of AV are key topics to achieving full vehicle autonomy where the driver becomes a passenger. It is necessary to give an idea of the complexity of the system since it involves perceiving and interpreting the environment, anticipating the actions of other road occupants to plan a safe trajectory, reacting to the unexpected, while respecting the regulations and ensuring the comfort and safety of passengers. The safety engineers lack the automatic interaction tools during the development of AV, which can help them understand the correctness of the safety rules, guarantee the reliability of the system in in-

```
Sat4jRulesConsistency.build_goal1_cond1_False(boolSpec);
Sat4jSystemConsistency.coherence_front_car_tracking(boolSpec);
```

(a) Functions tested in parallel to solve the inconsistency in the states of the same property.

```
Solution 12:
[-1, -2, -3, 4, -5, 6, -7, -8, -9, -10, 11, 12, -13, 14, -15]
1: goal1_cond1=0,
2: goal1_cond1_1=0,
3: goal1_cond1_2=0,
4: not_exist_front_car_distance=1,
5: goal1_cond1_1_1=0,
6: not_exist_front_car_distancegoal1_cond1_1_1=1,
7: not_exist_front_car_tracking=0,
8: disappeared_less_than_t1_front_car_tracking=0,
9: disappeared_more_than_t1_front_car_tracking=0,
10: not_exist_straddling_car_distance=0,
11: not_exist_straddling_car_tracking=1,
12: not_exist_straddling_car_distancenot_exist_straddling_car_tracking=1,
13: not_confirmed_front_car_tracking=0,
14: stable_tracking_front_car_tracking=1,
15: coherence_front_car_tracking=0,
```

(b) Solution for **goal1_cond1** that considers the coherence of **front_car_tracking** property.

Figure 6.54: Improving tests to eliminate inconsistent solutions considering properties coherence.

```
[[1, 2, -3, -4, -5]]
Solution 1:
[1, 2, -3, -4, -5]
1: goal4_cond2=1,
2: not_confirmed_front_car_tracking=1,
3: goal4_cond1=0,
4: imminent_collision_distance_front_car_distance=0,
5: imminent_collision_distance_straddling_car_distance=0,
```

Figure 6.55: Inconsistency found after a priority test for **goal1_cond1** and **goal2_cond1**.

```
Solution 6:
[1, 2, 3, -4, 5, 6, 7, 8]
1: goal2_cond1=1,
2: stable_stable_control=1,
3: goal2_cond1_1=1,
4: disappeared_more_than_t1_front_car_tracking=0,
5: no_detection_in_more_than_t6_line_detection=1,
6: straddling_more_than_t7_straddling_car_tracking=1,
7: goal3_cond1=1,
8: imminent_collision_distance_front_car_distance=1,
```

Figure 6.56: Inconsistency found after a parallel test for **goal3_cond1** and **goal2_cond1**.

numerable possible situations, and its resistance to possible malicious attacks. In this Chapter, we discussed the benefits of introducing a new workflow using

the EPSAAV tool which marked a new mode of interaction between the safety engineer and the development system for AVs. In Renault, some of the engineers are still using traditional methods while new research is being done using model-based approaches to the safety that have become prominent. The proposed approach tackles these problems by enhancing safety engineers to define safety requirements that follow one common format in the Safety domain. We show how this approach can reduce development costs and improve the correctness of safety rules, by enabling the direct interaction of safety engineers with a system under development. Additionally, this approach contributes to development by enabling quick reuse and improvements of existing solutions. The approach we are providing helps him save development time and cost by obtaining generated documents and monitors. He can check violations and inconsistencies in his requirements and takes full responsibility in the Safety domain. We report the *Area2Spec* use case on the assessment of the validation of the EPSAAV tool. *Area2Spec* consists of five scenarios that we specify formally and define requirements using these unsafe known scenarios. We show how we apply **EPSAAV specifier** for this case study and how we generate monitors and verification engine using **EPSAAV generator**. All the environment of *Area2Spec* is detailed and expressed using logical operators. We show how the developers can connect the code to the debugger, and how they can detect ambiguities. We show also how safety engineers can apply generated tests to validate their execution types and test inconsistencies by giving system solutions.

Chapter 7

Conclusion

7.1 Thesis summary

The main goal of this thesis is to propose a solution for assessing the safety requirements of AVs using DSML. We started by introducing our research objectives, formulating research questions, and presenting our research approach (Chapter 1). We continued by understanding the relevant safety domain, safety approaches and standards that ensure safety (Chapter 2). Furthermore, we also introduced in Chapter 2 MDE and DSML contexts (its implementation, stakeholders, and life-cycle) and highlighted a state of practice in safety assessment and identifying traditional safety techniques using MBSE and efforts which may lead to safety assessment. We illustrated how the EPSAAV can be integrated with DSML development and discussed the applicability of the EPSAAV approach to incremental iterative DSL development.

We proposed to approach our problem by introducing EPSAAV design model which is general enough to be applied to different safety studies (Chapter 3). This model was further placed in a process of DSML development which was defined as a pattern language. We captured all the relevant concepts and activities, by using an MDD approach, in the EPSAAV conceptual framework using Gemoc Studio. This framework groups the abstract syntax that defines the EPSAAV metamodel, a concrete syntax that enables engineers to inject rules and libraries using an interactive console. This latter constitutes the **EPSAAV specifier** to formalize specifications according to specific semantics. A generation of a document, a monitor, and a verification engine that correspond to specific studies and analysis is provided using **EPSAAV generator**.

One monitor generated in C language, called the Safety Checker module, is compatible with Renault's simulation environment named FusionRunner to detect ambiguities when violations occur. We detail the process in the following Chapter 4. FusionRunner performs resimulation and debugging of generated or real scenarios and presents many advantages such as calculating all necessary

information and improving algorithmic performances. It also gives us access to raw or fusion data. It generates metrics that can be consulted via an interface in order to analyze at a glance the quality and performance of the software and to quickly identify the problems introduced. This is why the choice fell on FusionRunner note that the monitor can adapt to the language chosen.

A verification engine is generated in Chapter 5 in which we have discussed the necessity of using a solver. The choice fell on the SAT solver (SAT4J library) to ensure results with simple Boolean operators. Note we can change the SAT and use the SMT for more complex systems since the source code generated is adaptable with any solver type. The objective of this contribution is to arrive at a solid and complete system that helps the engineer to integrate his safety requirements knowing that the inconsistencies will be dealt with. We presented the three implementation phases to deploy the SAT solver. This generation follows specific Boolean formulas. A set of Boolean operators are combined into these Boolean logic formulas that are fed to the SAT solver. We also detail in this Chapter the generation tests of Java code to study the inconsistencies between the rules and the system.

From the point of view of the feasibility of the EPSAAV conceptual framework to support the DSML for safety evaluation, the generated monitor and the verification engine were used in Chapter 6 to approve the usefulness of the previous contributions. We discussed the potential users of the EPSAAV approach. The prototype was used to instantiate the usability evaluation of industrial safety works. The final step was to prove that our approach provides a solution to our research problems by explaining the SAVI process with a case study. AREA2SPEC served to illustrate our solution design. Obviously, a lot of effort needs to be put into the infrastructure to support these features. However, it reduces the development time for the safety engineer who may not be an expert in programming.

7.2 Results obtained

The problem tackled in this thesis is very well-known in the safety area. To our knowledge before this thesis was written, there was no real attempt to tackle the problem in such a global and methodical manner. Therefore, during our thesis argumentation, we believe to have introduced the systematic approach which promotes quality in the use of DSLs, during their development process by leveraging usability as a first-class concern. We present a summary of the main results achieved and some of the benefits of the development of this dissertation:

1. **To define an appropriate level of abstraction for safety evaluation, we developed a systematic approach for safety evaluation of AVs environment. We proposed a conceptual modeling framework called EPSAAV that regroups specifier and generators**. This comprehensive framework which is presented in Chapter 3 identifies all the

mandatory concepts and activities and aggregates them into a formal meta-model. It highlights the complexity of the information or specifications that should be traced to streamline and automate the generation process. The conceptual framework contributed directly to the thesis objective, by providing a set of practices that the safety engineer should follow to provide a complete solution to a safety problem. The framework helps the safety engineers to explicitly model the evaluation process, which contributes to formalizing their requirements according to ASIL levels. We applied our approach in a real-life case study of the usability evaluation of known unsafe scenarios for SOTIF requirements in Chapter 6.

2. **To integrate a tool to help an automatic generation of documents and monitors that solves safety assessment problems, we have developed a tool that provides an automation scheme to help generate easy-to-use monitors and engines**. This generation bridges the gap between safety experts and developers from the definition of safety requirements to the validation of these specifications. Safety experts do not need to have expertise in development to assess safety in real or simulated scenarios.

3. **To promote detection of violations to help the safety engineer catch violations and ambiguities in his defined requirements, we proposed to provide an approach that associates the EPSAAV framework to a simulator seen in Chapter 4**. The problem here is that the safety experts are not sometimes involved in the language development and may not be the end-users. They may miss a specification that is important for failure detection, which leads to an immature safety evaluation. They may therefore introduce biases in the perception of the language design and its usability. We proposed a systematic approach that generates a monitor that is fed to the simulator, in our case study we used a debugger called FusionRunner that does resimulation of real or simulated scenarios. We have applied our approach in a real case study of violations detection and evaluation using FusionRunner in Chapter 6.

4. **In order to favor the detection of rules inconsistencies to help the safety engineer to revise and reconsider his requirements, we proposed to provide an approach that combines the EPSAAV framework with a solver to find all possible solutions in Chapter 5**. We chose the SAT solver to study the logical expressiveness of the rules and run priority and parallel tests on its rules to ensure that no inconsistencies exist. We have applied our approach in a real case study of inconsistencies detection and evaluation using SAT solver in Chapter 6.

Some of the benefits arising from these results and confirmed by case studies:

- **Feasibility of the EPSAAV conceptual framework to support the safety engineers to prepare safety evaluation**. The case study provided valuable feedback and improvements suggestions over the framework. The framework does not only help the safety experts follow a specific format of describing the specifications using the EPSAAV specifier, but also eliminates syntactical errors that could be produced in the description. Further, the framework gives him a document to validate the correctness of his specifications, a monitor to verify and check violations, and a formal verification engine to analyze rules inconsistencies. All these generated files are provided by the EPSAAV generator. Finally, we conducted an evaluation study of the traditional workflow in Renault, and the workflow using the proposed approach which confirmed the feasibility of the EPSAAV conceptual framework to support safety engineers in conducting safety evaluations.

- **Traceability of safety assessment**. Our systematic approach enabled tracing a justification of the safety assessment, and its impact throughout the evolution of the DSML. Each safety evaluation is a concrete instantiation of a high-level safety objective. It refers to the precise context which stores the assumptions which impacted the safety evaluation decision. As a consequence of the evaluation, new features get discovered, and development priorities can change according to tests and analyses.

- **Easy integration of the monitor and the verification engine and reduction in time development**. The given systematic approach for file generation does not depend on any particular technology. The developer only needs to connect the files with the debugger or simulator and give them back to the safety engineer. If the technologies change in Renault, we can adapt the generation without losing time development. It was specified taking into consideration the reuse of the existing knowledge enabling easy integration with existing artefacts or assessment support.

- **Design of reusable safety experiments**. Safety assessment modeling defines objects of experience that are reusable. We have shown in the case of AREA2SPEC (Section 6.8) that the tool gives the possibility to modify the requirements and to benefit from this reuse by having a comparison between the current and previous versions of the rule-based planner.

7.3 Future work

In this Section, we are pointing out the future work which emerged from the proposed solution. Namely, we highlight the following two work directions to be explored:

7.3.1 Evolution of the EPSAAV tool

The EPSAAV approach, besides documenting the safety evaluation process, enables formalizing the requirements and assessing rule violations and inconsistencies.
As part of the future work, we are improving the tool to support time constraints to test safety after a certain time, such as *execute Action after n seconds*. This could be implemented in Gemoc Studio using Clock Constraint Specification Language (CCSL) [142] to achieve a model Checking. CCSL gives the possibility to better express temporal information such as a state preceding the other, and helps to study concurrent behaviors. Another improvement could be done on the definition of the requirement that could be taken into account graphically instead of syntactically. This could also be implemented in Gemoc Studio using Sirius [143].
In future work, we can also improve the resolution of solutions and filter them by applying more automatically generated tests for the solver such as applying a coherence test on behaviors shared by the same requirements. Furthermore, we can create methods to indicate the inconsistency to the safety engineer instead of looking at all the solutions. We can also choose the SMT solver for more complex systems since we have proved that the SAT solver works perfectly and demonstrated that we can have solutions according to the specified requirements. Last, we can add architect engineers in the workflow which communicates between the safety engineer and the development engineer, and discuss future works such as adding new simulators or using other solvers.

7.3.2 Reusability of the EPSAAV tool in various contexts

We underlined the usability of the EPSAAV tool in the field of safety. More testing for existing or simulated scenarios could be done using FusionRunner to visualize more violations and detect ambiguities. The current tests were performed under five scenarios following the SOTIF standard for known hazardous scenarios. The goal is to provide feature safety for unforeseen scenarios that might be encountered by the system. We can consider for future work applying the proposed EPSAAV tool on electrical and electronic malfunctions since SOTIF supplements ISO 26262. It uses the same vocabulary but extends it with standalone specific terms, and its scope is complementary to ISO 26262. They are different standards, and it is their combination that helps autonomous developers avoid dangerous situations, both in the presence and absence of malfunctions and unintended use cases.
In this thesis, we dug into the ADAS component to create a safety module checker for AEB and ACC components (see Figure 7.1). If an inconsistency exists between the output signals from the ECUs and the Safety Module Checker, the latter can make the final decision. For future work, applying safety requirements

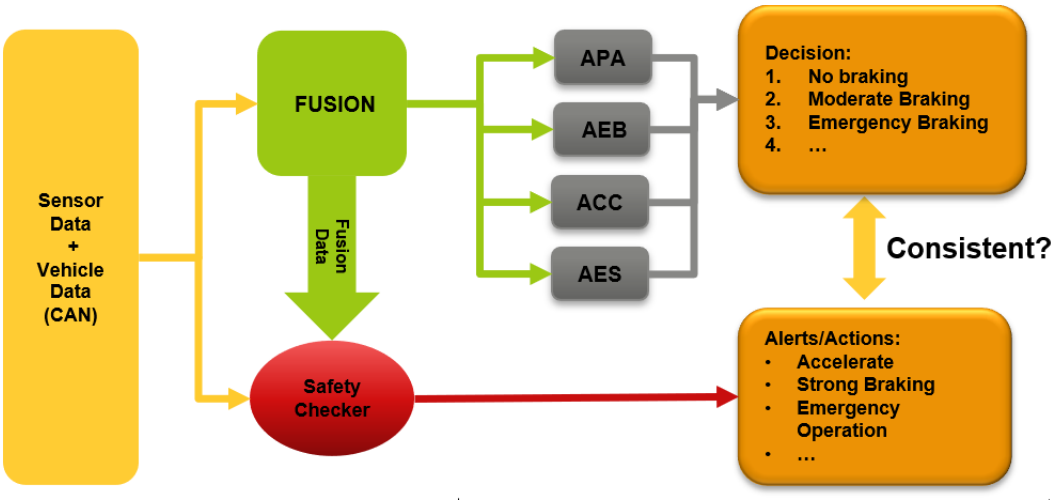

Figure 7.1: Safety Checker Module replacing ADAS functionality blocks.

on other functional blocks such as APA, AES, and other ECUs could be a possible solution to separate safety from path planning and gives the Safety domain hand in a decision if an inconsistency occurs.
In addition, we find that reusing and reasoning the tool can also be reused in various domains such as security and comfortability. To make this possible, it is necessary to obtain an appropriate number of specifications, and organize them with the support of experts. We leave this for future work as it is not the focus of this thesis.

Abbreviations

ACC	Adaptive Cruise Control
AEB	Autonomous Emergency Braking
AES	Automatic Emergency Steering
AD	Autonomous Driving
APA	Advanced Park Assist
ASIL	Automotive Safety Integrity Levels
AV	Autonomous Vehicle
CCSL	Clock Constraint Specification Language
CPU	Central Processing Unit
DSL	Domain Specific Language
DSML	Domain-Specific Modeling Language
ECU	Electronic Control Unit
EMF	Eclipse Modeling Framework
EPSAAV	Extensible Platform for Safety Analysis of AVs
FMEA	Failure Modes and Effects Analysis
FV	Following Vehicle
FTA	Fault Tree Analysis
GPS	Global Positioning System
GUI	Graphical User Interface
HMI	Human-Machine Interface
IBM	International Business Machines
IMU	Inertial Measurement Unit
LKA	Lane Keeping Assistance
MBSE	Model-Based System Engineering
MDA	Model-Driven Architecture
MDD	Model-Driven Development
MDE	Model-Driven Engineering
ODD	Operational Design Domain
OEDR	Object and Event Detection and Response
OMG	Object Management Group
OO	Object Oriented
PV	Preceding Vehicle
RBP	Rule-Based Planner

SAE	Society of Automotive Engineers
SAT	Boolean Satisfiability
SAVI	Safety Analyses of Violations and Inconsistencies
SIL	Safety Integrity Levels
SMT	Satisfiability Modulo Theories
SV	Straddling Vehicle
TTC	Time To Collision
UML	Unified Modeling Language
VRU	Vehicle Road User

Appendix A

A.1 Model instantiation of requirements and environment

Listing A.1: Area2Ac.actions

```
//this is the definition of the library for the actions

ActionLibrary Area2Actions :
Action "brake_acc";
Action "brake_strong";
Action "report_longitudinal_delta_V";
Action "report_lateral_delta_V";
Action "emergency_operation_1";
Action "emergency_maneuver";
Action "bug_notification";
Action "no_action_needed";
```

Listing A.2: Area2Al.alerts

```
//this is the definition of the library for the alerts

AlertLibrary Area2Alerts :
Alert "longitudinal_acceleration";
Alert "longitudinal_information_reporter";
Alert "lateral_information_reporter";
Alert "bug_notification";
Alert "eop1";
Alert "no_alert_needed";
```

Listing A.3: Area2Ego.prop

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


PropertyTypeLibrary EgoProperties
```

```
{   version '1'
  state "traffic_jam" CanBe "yes" "no"
  state "front_car_distance" CanBe "not_exist"
  "exist"
  "safe_distance" operator">=" value 2.0 unit "s"
  "acc_distance" operator">=" value 1.2 unit "s"
  "strong_braking_distance" operator">=" value 0.8 unit "s"
  "imminent_collision_distance"
  state "front_car_tracking" CanBe "not_confirmed"
  "not_exist" operator">=" value 0.33 unit "s"
  "disappeared_less_than_t1" operator"<" value 0.25 unit "s"
  "disappeared_more_than_t1" operator">=" value 0.25 unit "s"
  "stable_tracking" operator">" value 0.25 unit "s"
  state "line_detection" CanBe "stable"
  "no_detection_in_less_than_t6" operator"<" value 0.3 unit "s"
  "no_detection_in_more_than_t6" operator">" value 0.3 unit "s"
  state "straddling_car_distance" CanBe "
     imminent_collision_distance"
  "safe_distance" operator"=" value 2.0 unit "s"
  "acc_distance" operator">=" value 1.2 unit "s"
  "strong_braking_distance" operator">=" value 0.8 unit "s"
  "not_exist" operator">=" value 0.33 unit "s"
  state "straddling_car_tracking" CanBe "not_exist"
  "straddling_less_than_t7" operator"<" value 2.0 unit "s"
  "straddling_more_than_t7" operator">=" value 2.0 unit "s"
  state "stable_control" CanBe "stable" "not_stable"
  state "rear_car_distance" CanBe "not_exist"
  "safe_distance" operator"=" value 2.0 unit "s"
  "emergency_distance"


}
```

Listing A.4: Area2Obj.otp

```
//this is the definition of the library for the Ego and Obstacle
   parameters

ObjectTypeLibrary OTL version'1':

Ego {
  EgoPropertyLibrary EgoProperties;
  EgoParameterLibrary Ego :
    parameter Lane_detection Type float;
    parameter Lane_missing Type float;
}

"Obstacle"{
  ParameterLibrary Obstacle :
    parameter id Type int32;
    parameter counter_tracking Type float;
```

```
    parameter counter_not_tracking Type float;
    parameter ttc_min Type float;
}
```

Listing A.5: Area2Sc.scene

```
//this is the definition of the scene

scene Scene {
  ego ^Ego
  role Ego objectType "OTL.Ego"
  role "preceding_vehicle" objectType "OTL.Obstacle"
  role "straddling_vehicle" objectType "OTL.Obstacle"
}
```

Listing A.6: Area2Spec.rbp

```
//this is the definition of the Rule Based Planner


RuleBasedPlanner RBP{
  scene ^Scene
  GOAL goal1
  {
  GoalType Priority
  WHEN{
    NOT(
      OR( //no_interference_in_front_car_detection :
      {   propertyType "EgoProperties.front_car_distance" is
        "EgoProperties.front_car_distance.not_exist"
        ifonlyif(
          OR (propertyType "EgoProperties.front_car_tracking" is
            "EgoProperties.front_car_tracking.not_exist",
            propertyType "EgoProperties.front_car_tracking" is
            "EgoProperties.front_car_tracking.
               disappeared_less_than_t1",
            propertyType "EgoProperties.front_car_tracking" is
            "EgoProperties.front_car_tracking.
               disappeared_more_than_t1"
          )
        )
      },  //no_interference_in_straddling_car_detection :
      {     propertyType "EgoProperties.straddling_car_distance"
         is
          "EgoProperties.straddling_car_distance.not_exist"
       ifonlyif (
          propertyType "EgoProperties.straddling_car_tracking" is
          "EgoProperties.straddling_car_tracking.not_exist"
          )
      }
```

```
    )
  )
  action "Area2Actions.bug_notification"
  alert "Area2Alerts.bug_notification"
}
  GOAL goal2{
    GoalType Constraint
    WHEN{
      AND(//no lane or infrastructure detection
      propertyType "EgoProperties.stable_control" is
        "EgoProperties.stable_control.stable",
        AND(NOT(propertyType "EgoProperties.line_detection" is
            "EgoProperties.line_detection.stable"),
        OR( propertyType "EgoProperties.front_car_tracking" is
        "EgoProperties.front_car_tracking.
            disappeared_more_than_t1",
          propertyType "EgoProperties.line_detection" is
          "EgoProperties.line_detection.
             no_detection_in_more_than_t6",
          //swerving SV to EV's lane
          propertyType "EgoProperties.straddling_car_tracking"
             is
          "EgoProperties.straddling_car_tracking.
             straddling_more_than_t7"
        ))
      )
      action "Area2Actions.emergency_operation_1"
      alert "Area2Alerts.eop1"
    }
    GOAL goal3{
    GoalType Constraint //report Delta V when imminent
       collision
    WHEN{
      propertyType "EgoProperties.front_car_distance" is
      "EgoProperties.front_car_distance.
         imminent_collision_distance"
      action "Area2Actions.report_longitudinal_delta_V"
      alert "Area2Alerts.longitudinal_acceleration"
    }
    WHEN{
      propertyType "EgoProperties.straddling_car_distance" is
      "EgoProperties.straddling_car_distance.
         imminent_collision_distance"
      action "Area2Actions.report_longitudinal_delta_V"
      alert "Area2Alerts.longitudinal_acceleration"
    }
      GOAL goal4{
      GoalType Priority
        WHEN{//imminent collision
          OR(propertyType "EgoProperties.front_car_distance" is
             "EgoProperties.front_car_distance.
```

```
                imminent_collision_distance",
              propertyType "EgoProperties.
                 straddling_car_distance" is
           "EgoProperties.straddling_car_distance.
              imminent_collision_distance")
        action "Area2Actions.emergency_maneuver"
        alert "Area2Alerts.longitudinal_acceleration"
      }
      WHEN{//during confirmation of PV
        propertyType "EgoProperties.front_car_tracking" is
        "EgoProperties.front_car_tracking.not_confirmed"
        action "Area2Actions.brake_acc"
        alert "Area2Alerts.longitudinal_acceleration"
      }
      WHEN{//collision with the rear
        propertyType "EgoProperties.rear_car_distance" is
        "EgoProperties.rear_car_distance.emergency_distance"
        action "Area2Actions.no_action_needed"
        alert "Area2Alerts.no_alert_needed"
      }
      WHEN{//strong braking collision
        OR(propertyType "EgoProperties.front_car_distance" is
           "EgoProperties.front_car_distance.
              strong_braking_distance",
           propertyType "EgoProperties.
              straddling_car_distance" is
           "EgoProperties.straddling_car_distance.
              strong_braking_distance")
        action "Area2Actions.brake_strong"
        alert "Area2Alerts.longitudinal_acceleration"
      }
      WHEN{//acc collision
        OR(propertyType "EgoProperties.front_car_distance" is
           "EgoProperties.front_car_distance.acc_distance",
           propertyType "EgoProperties.
              straddling_car_distance" is
           "EgoProperties.straddling_car_distance.
              acc_distance")
        action "Area2Actions.brake_acc"
        alert "Area2Alerts.longitudinal_acceleration"
      }
    }
  }
 }
}
```

A.2 Text auto-generation for *Area2Spec*

Listing A.7: Area2Spec.txt

```
BEGIN GOAL goal1{
  Goal type: Priority
  when{
      NOT (
        front_car_distance is not_exist
        IFONLYIF(
          front_car_tracking is not_exist
          OR(
            front_car_tracking is disappeared_less_than_t1,
            front_car_tracking is disappeared_more_than_t1
          )
        )
        OR(
          straddling_car_distance is not_exist
          IFONLYIF(
            straddling_car_tracking is not_exist
          )
        )
      )
  }
  then behavior_goal1_cond1:
    executing bug_notification
    bug_notification

  BEGIN GOAL goal2{
    Goal type: Constraint
    when{
        stable_control is stable
        AND(
          NOT (
            line_detection is stable
          )
          AND(
            front_car_tracking is disappeared_more_than_t1
            OR(
              line_detection is no_detection_in_more_than_t6,
              straddling_car_tracking is straddling_more_than_t7
            )
          )
        )
    }
    then behavior_goal2_cond1:
      executing emergency_operation_1
      eop1

    BEGIN GOAL goal3{
      Goal type: Constraint
      when{
          front_car_distance is imminent_collision_distance
      }
      then behavior_goal3_cond1:
```

```
  executing report_longitudinal_delta_V
  longitudinal_acceleration

when{
    straddling_car_distance is imminent_collision_distance
}
then behavior_goal3_cond2:
  executing report_longitudinal_delta_V
  longitudinal_acceleration

BEGIN GOAL goal4{
  Goal type: Priority
  when{
      front_car_distance is imminent_collision_distance
      OR(
        straddling_car_distance is
            imminent_collision_distance
      )
  }
  then behavior_goal4_cond1:
    executing emergency_maneuver
    longitudinal_acceleration

  when{
      front_car_tracking is not_confirmed
  }
  then behavior_goal4_cond2:
    executing brake_acc
    longitudinal_acceleration

  when{
      rear_car_distance is emergency_distance
  }
  then behavior_goal4_cond3:
    executing no_action_needed
    no_alert_needed

  when{
      front_car_distance is strong_braking_distance
      OR(
        straddling_car_distance is strong_braking_distance
      )
  }
  then behavior_goal4_cond4:
    executing brake_strong
    longitudinal_acceleration

  when{
      front_car_distance is acc_distance
      OR(
        straddling_car_distance is acc_distance
```

```
            )
        }
        then behavior_goal4_cond5:
          executing brake_acc
          longitudinal_acceleration

      }
    }
  }
}
BEGIN RESOURCES
  resource longitudinal_acceleration
  resource longitudinal_information_reporter
  resource lateral_information_reporter
  resource bug_notification
  resource eop1
  resource no_alert_needed
END RESOURCES

BEGIN ACTIONS
      action brake_acc
      action brake_strong
      action report_longitudinal_delta_V
      action report_lateral_delta_V
      action emergency_operation_1
      action emergency_maneuver
      action bug_notification
      action no_action_needed
    END ACTIONS

BEGIN STATE VECTOR
PropertyTypeLibrary EgoProperties{
    state traffic_jam CanBe  yes  no
    state front_car_distance CanBe  not_exist  exist
       safe_distance  acc_distance  strong_braking_distance
       imminent_collision_distance
    state front_car_tracking CanBe  not_confirmed  not_exist
       disappeared_less_than_t1  disappeared_more_than_t1
       stable_tracking
    state line_detection CanBe  stable
       no_detection_in_less_than_t6  no_detection_in_more_than_t6
    state straddling_car_distance CanBe
       imminent_collision_distance  safe_distance  acc_distance
       strong_braking_distance  not_exist
    state straddling_car_tracking CanBe  not_exist
       straddling_less_than_t7  straddling_more_than_t7
    state stable_control CanBe  stable  not_stable
    state rear_car_distance CanBe  not_exist  safe_distance
       emergency_distance
}
END STATE VECTOR
```

A.3 C code auto-generation using EPSAAV generator

Listing A.8: Area2Actions.c

```
/** \file Area2Actions.c
*
* \author
* \brief action library functions for safety checker
*
*/

#include "Area2Actions.h"
#include <stdio.h>

void processactions(Area2Actions_t Area2Actions_enum) {
    switch (Area2Actions_enum) {
      case ac_brake_acc:
        printf("brake_acc\n");
        break;
      case ac_brake_strong:
        printf("brake_strong\n");
        break;
      case ac_report_longitudinal_delta_V:
        printf("report_longitudinal_delta_V\n");
        break;
      case ac_report_lateral_delta_V:
        printf("report_lateral_delta_V\n");
        break;
      case ac_emergency_operation_1:
        printf("emergency_operation_1\n");
        break;
      case ac_emergency_maneuver:
        printf("emergency_maneuver\n");
        break;
      case ac_bug_notification:
        printf("bug_notification\n");
        break;
      case ac_no_action_needed:
        printf("no_action_needed\n");
        break;
      default:
        printf("no_action\n");
        break;
    }
}
```

Listing A.9: Area2Actions.h

```
/** \file Area2Actions.h
```

```
*
* \author
* \brief action library for safety checker
*
*/
#ifndef Area2Actions_H_
#define Area2Actions_H_

typedef enum {
  ac_brake_acc,
  ac_brake_strong,
  ac_report_longitudinal_delta_V,
  ac_report_lateral_delta_V,
  ac_emergency_operation_1,
  ac_emergency_maneuver,
  ac_bug_notification,
  ac_no_action_needed,
} Area2Actions_t;

void processactions(Area2Actions_t Area2Actions_enum);

#endif
```

Listing A.10: Area2Alerts.c

```
/** \file Area2Alerts.c
*
* \author
* \brief alert library functions for safety checker
*
*/

#include "Area2Alerts.h"
#include <stdio.h>

void processalerts(Area2Alerts_t Area2Alerts_enum) {
    switch (Area2Alerts_enum) {
    case al_longitudinal_acceleration:
      printf("longitudinal_acceleration\n");
      break;
    case al_longitudinal_information_reporter:
      printf("longitudinal_information_reporter\n");
      break;
    case al_lateral_information_reporter:
      printf("lateral_information_reporter\n");
      break;
    case al_bug_notification:
      printf("bug_notification\n");
      break;
    case al_eop1:
      printf("eop1\n");
```

```
      break;
    case al_no_alert_needed:
      printf("no_alert_needed\n");
      break;
    default:
      printf("no_alert\n");
      break;
    }
}
```

Listing A.11: Area2Alerts.h

```
/** \file Area2Alerts.h
*
* \author
* \brief alert library for safety checker
*
*/
#ifndef Area2Alerts_H_
#define Area2Alerts_H_

typedef enum {
  al_longitudinal_acceleration,
  al_longitudinal_information_reporter,
  al_lateral_information_reporter,
  al_bug_notification,
  al_eop1,
  al_no_alert_needed,
} Area2Alerts_t;

void processalerts(Area2Alerts_t Area2Alerts_enum);

#endif
```

Listing A.12: EgoProperties.c

```
/** \file EgoProperties.c
*
* \author
* \brief EgoProperties functions for safety checker
*
*/

#include "EgoProperties.h"
#include "Scene.h"


traffic_jam_t traffic_jam_check() {

  /*TODO User Adaptation : functions declared in Scene.c with
     thresholds
```

```
  if () {
    return yes_traffic_jam;
  } else {
    return no_traffic_jam;
  }
  */
  return yes_traffic_jam;
}


front_car_distance_t front_car_distance_check() {

  /*TODO User Adaptation : functions declared in Scene.c with
     thresholds
  static const float safe_distance_threshold = 2.0f; //
     safe_distance>=2.0s
  static const float acc_distance_threshold = 1.2f; //
     acc_distance>=1.2s
  static const float strong_braking_distance_threshold = 0.8f; //
     strong_braking_distance>=0.8s

  if () {
    return not_exist_front_car_distance;
  }else if () {
    return exist_front_car_distance;
  }else if () {
    return safe_distance_front_car_distance;
  }else if () {
    return acc_distance_front_car_distance;
  }else if () {
    return strong_braking_distance_front_car_distance;
  } else {
    return imminent_collision_distance_front_car_distance;
  }
  */
  return not_exist_front_car_distance;
}


front_car_tracking_t front_car_tracking_check() {

  /*TODO User Adaptation : functions declared in Scene.c with
     thresholds
  static const float not_exist_threshold = 0.33f; //not_exist
     >=0.33s
  static const float disappeared_less_than_t1_threshold = 0.25f;
     //disappeared_less_than_t1<0.25s
  static const float disappeared_more_than_t1_threshold = 0.25f;
     //disappeared_more_than_t1>=0.25s
  static const float stable_tracking_threshold = 0.25f; //
```

```
      stable_tracking>0.25s

  if () {
    return not_confirmed_front_car_tracking;
  }else if () {
    return not_exist_front_car_tracking;
  }else if () {
    return disappeared_less_than_t1_front_car_tracking;
  }else if () {
    return disappeared_more_than_t1_front_car_tracking;
  } else {
    return stable_tracking_front_car_tracking;
  }
  */
  return not_confirmed_front_car_tracking;
}


line_detection_t line_detection_check() {

  /*TODO User Adaptation : functions declared in Scene.c with
      thresholds
  static const float no_detection_in_less_than_t6_threshold = 0.3
      f; //no_detection_in_less_than_t6<0.3s
  static const float no_detection_in_more_than_t6_threshold = 0.3
      f; //no_detection_in_more_than_t6>0.3s

  if () {
    return stable_line_detection;
  }else if () {
    return no_detection_in_less_than_t6_line_detection;
  } else {
    return no_detection_in_more_than_t6_line_detection;
  }
  */
  return stable_line_detection;
}


straddling_car_distance_t straddling_car_distance_check() {

  /*TODO User Adaptation : functions declared in Scene.c with
      thresholds
  static const float safe_distance_threshold = 2.0f; //
      safe_distance=2.0s
  static const float acc_distance_threshold = 1.2f; //
      acc_distance>=1.2s
  static const float strong_braking_distance_threshold = 0.8f; //
      strong_braking_distance>=0.8s
  static const float not_exist_threshold = 0.33f; //not_exist
      >=0.33s
```

```
  if () {
    return imminent_collision_distance_straddling_car_distance;
  }else if () {
    return safe_distance_straddling_car_distance;
  }else if () {
    return acc_distance_straddling_car_distance;
  }else if () {
    return strong_braking_distance_straddling_car_distance;
  } else {
    return not_exist_straddling_car_distance;
  }
  */
  return imminent_collision_distance_straddling_car_distance;
}


straddling_car_tracking_t straddling_car_tracking_check() {

  /*TODO User Adaptation : functions declared in Scene.c with
     thresholds
  static const float straddling_less_than_t7_threshold = 2.0f; //
     straddling_less_than_t7<2.0s
  static const float straddling_more_than_t7_threshold = 2.0f; //
     straddling_more_than_t7>=2.0s

  if () {
    return not_exist_straddling_car_tracking;
  }else if () {
    return straddling_less_than_t7_straddling_car_tracking;
  } else {
    return straddling_more_than_t7_straddling_car_tracking;
  }
  */
  return not_exist_straddling_car_tracking;
}


stable_control_t stable_control_check() {

  /*TODO User Adaptation : functions declared in Scene.c with
     thresholds

  if () {
    return stable_stable_control;
  } else {
    return not_stable_stable_control;
  }
  */
  return stable_stable_control;
}
```

```
rear_car_distance_t rear_car_distance_check() {

  /*TODO User Adaptation : functions declared in Scene.c with
     thresholds
  static const float safe_distance_threshold = 2.0f; //
     safe_distance=2.0s

  if () {
    return not_exist_rear_car_distance;
  }else if () {
    return safe_distance_rear_car_distance;
  } else {
    return emergency_distance_rear_car_distance;
  }
  */
  return not_exist_rear_car_distance;
}
```

Listing A.13: EgoProperties.h

```
/** \file EgoProperties.h
*
* \author
* \brief Ego Property library for safety checker
*
*/

#include <math.h>
#ifndef EgoProperties_H_
#define EgoProperties_H_

typedef enum {
  yes_traffic_jam,
  no_traffic_jam,
} traffic_jam_t;

typedef enum {
  not_exist_front_car_distance,
  exist_front_car_distance,
  safe_distance_front_car_distance,
  acc_distance_front_car_distance,
  strong_braking_distance_front_car_distance,
  imminent_collision_distance_front_car_distance,
} front_car_distance_t;

typedef enum {
  not_confirmed_front_car_tracking,
  not_exist_front_car_tracking,
  disappeared_less_than_t1_front_car_tracking,
```

```
  disappeared_more_than_t1_front_car_tracking,
  stable_tracking_front_car_tracking,
} front_car_tracking_t;

typedef enum {
  stable_line_detection,
  no_detection_in_less_than_t6_line_detection,
  no_detection_in_more_than_t6_line_detection,
} line_detection_t;

typedef enum {
  imminent_collision_distance_straddling_car_distance,
  safe_distance_straddling_car_distance,
  acc_distance_straddling_car_distance,
  strong_braking_distance_straddling_car_distance,
  not_exist_straddling_car_distance,
} straddling_car_distance_t;

typedef enum {
  not_exist_straddling_car_tracking,
  straddling_less_than_t7_straddling_car_tracking,
  straddling_more_than_t7_straddling_car_tracking,
} straddling_car_tracking_t;

typedef enum {
  stable_stable_control,
  not_stable_stable_control,
} stable_control_t;

typedef enum {
  not_exist_rear_car_distance,
  safe_distance_rear_car_distance,
  emergency_distance_rear_car_distance,
} rear_car_distance_t;

traffic_jam_t traffic_jam_check();

front_car_distance_t front_car_distance_check();

front_car_tracking_t front_car_tracking_check();

line_detection_t line_detection_check();

straddling_car_distance_t straddling_car_distance_check();

straddling_car_tracking_t straddling_car_tracking_check();

stable_control_t stable_control_check();

rear_car_distance_t rear_car_distance_check();
```

```
#endif
```

Listing A.14: Scene.c

```
/** \file Scene.c
*
* \author
* \brief scene for safety checker
*
*/

#include "Scene.h"


static Ego_t Ego;
static Obstacle_t preceding_vehicle;
static Obstacle_t straddling_vehicle;

static void Ego_getParameters(/*TODO User Adaptation : Parameter
    FusionControlData */);
static void preceding_vehicle_getParameters(/*TODO User
    Adaptation : Parameter FusionControlData */);
static void straddling_vehicle_getParameters(/*TODO User
    Adaptation : Parameter FusionControlData */);

void fusionDatatoScene(/*TODO User Adaptation : Parameter
    FusionControlData */) {
  Ego_getParameters(/*TODO User Adaptation : Parameter
      FusionControlData */);
  preceding_vehicle_getParameters(/*TODO User Adaptation :
      Parameter FusionControlData */);
  straddling_vehicle_getParameters(/*TODO User Adaptation :
      Parameter FusionControlData */);
}

static void Ego_getParameters(/*TODO User Adaptation : Parameter
    FusionControlData */) {
  /*TODO User Adaptation : Link Fusion Output with Autogenerated
      Parameters in Scene.h */
}

static void preceding_vehicle_getParameters(/*TODO User
    Adaptation : Parameter FusionControlData */) {
  /*TODO User Adaptation : Link Fusion Output with Autogenerated
      Parameters in Scene.h */
}

static void straddling_vehicle_getParameters(/*TODO User
    Adaptation : Parameter FusionControlData */) {
  /*TODO User Adaptation : Link Fusion Output with Autogenerated
```

```
      Parameters in Scene.h */
}


float_t get_Ego_Lane_detection() {
  return Ego.Lane_detection;
}

float_t get_Ego_Lane_missing() {
  return Ego.Lane_missing;
}

uint32_t get_preceding_vehicle_id() {
  return preceding_vehicle.id;
}

float_t get_preceding_vehicle_counter_tracking() {
  return preceding_vehicle.counter_tracking;
}

float_t get_preceding_vehicle_counter_not_tracking() {
  return preceding_vehicle.counter_not_tracking;
}

float_t get_preceding_vehicle_ttc_min() {
  return preceding_vehicle.ttc_min;
}

uint32_t get_straddling_vehicle_id() {
  return straddling_vehicle.id;
}

float_t get_straddling_vehicle_counter_tracking() {
  return straddling_vehicle.counter_tracking;
}

float_t get_straddling_vehicle_counter_not_tracking() {
  return straddling_vehicle.counter_not_tracking;
}

float_t get_straddling_vehicle_ttc_min() {
  return straddling_vehicle.ttc_min;
}
```

Listing A.15: Scene.h

```
/** \file Scene.h
*
* \author
* \brief scene struct for safety checker
*
```

```
*/

#include "../src/fusrun_types.h"
#ifndef Scene_H_
#define Scene_H_

//functions instantiation:

void fusionDatatoScene(/*TODO User Adaptation : Parameter
   FusionControlData */);

float_t get_Ego_Lane_detection();
float_t get_Ego_Lane_missing();

uint32_t get_preceding_vehicle_id();
float_t get_preceding_vehicle_counter_tracking();
float_t get_preceding_vehicle_counter_not_tracking();
float_t get_preceding_vehicle_ttc_min();

uint32_t get_straddling_vehicle_id();
float_t get_straddling_vehicle_counter_tracking();
float_t get_straddling_vehicle_counter_not_tracking();
float_t get_straddling_vehicle_ttc_min();


//structures instantiation:

typedef struct Ego {
  float_t Lane_detection;
  float_t Lane_missing;
} Ego_t;

typedef struct Obstacle {
  uint32_t id;
  float_t counter_tracking;
  float_t counter_not_tracking;
  float_t ttc_min;
} Obstacle_t;

#endif
```

Listing A.16: Safety_Checks.c

```
#include "EgoProperties.h"
#include "Area2Actions.h"
#include "Area2Alerts.h"
#include "Scene.h"
#include "Safety_Checks.h"
#include <stdio.h>
/** \file Safety_Checks.c
*
```

```
* \author
* \brief safety checker
*
*/


void init_goals() {
goal1_condition1.isTriggered = &trig_goal1_condition1;
goal1_condition1.execute = &execute_goal1_condition1;
goal1_condition1.raiseAlarm = &alarm_goal1_condition1;

goal2_condition1.isTriggered = &trig_goal2_condition1;
goal2_condition1.execute = &execute_goal2_condition1;
goal2_condition1.raiseAlarm = &alarm_goal2_condition1;

goal3_condition1.isTriggered = &trig_goal3_condition1;
goal3_condition1.execute = &execute_goal3_condition1;
goal3_condition1.raiseAlarm = &alarm_goal3_condition1;

goal3_condition2.isTriggered = &trig_goal3_condition2;
goal3_condition2.execute = &execute_goal3_condition2;
goal3_condition2.raiseAlarm = &alarm_goal3_condition2;

goal4_condition1.isTriggered = &trig_goal4_condition1;
goal4_condition1.execute = &execute_goal4_condition1;
goal4_condition1.raiseAlarm = &alarm_goal4_condition1;

goal4_condition2.isTriggered = &trig_goal4_condition2;
goal4_condition2.execute = &execute_goal4_condition2;
goal4_condition2.raiseAlarm = &alarm_goal4_condition2;

goal4_condition3.isTriggered = &trig_goal4_condition3;
goal4_condition3.execute = &execute_goal4_condition3;
goal4_condition3.raiseAlarm = &alarm_goal4_condition3;

goal4_condition4.isTriggered = &trig_goal4_condition4;
goal4_condition4.execute = &execute_goal4_condition4;
goal4_condition4.raiseAlarm = &alarm_goal4_condition4;

goal4_condition5.isTriggered = &trig_goal4_condition5;
goal4_condition5.execute = &execute_goal4_condition5;
goal4_condition5.raiseAlarm = &alarm_goal4_condition5;

}

void trig_goals() {
  if ( goal1_condition1.isTriggered() == FALSE ) {
    goal2_condition1.isTriggered();
    goal3_condition1.isTriggered();
    goal3_condition2.isTriggered();
    if ( goal4_condition1.isTriggered() == FALSE ) {
```

```
      if ( goal4_condition2.isTriggered() == FALSE ) {
        if ( goal4_condition3.isTriggered() == FALSE ) {
          if ( goal4_condition4.isTriggered() == FALSE ) {
            goal4_condition5.isTriggered();
          }
        }
      }
    }
  }
}

bool_t trig_goal1_condition1() {
  if ( !( ( ( front_car_distance_check() ==
      not_exist_front_car_distance ) && ( (
      front_car_tracking_check() == not_exist_front_car_tracking )
       || ( front_car_tracking_check() ==
      disappeared_less_than_t1_front_car_tracking ) || (
      front_car_tracking_check() ==
      disappeared_more_than_t1_front_car_tracking )  )  ) || ( (
      straddling_car_distance_check() ==
      not_exist_straddling_car_distance ) && (
      straddling_car_tracking_check() ==
      not_exist_straddling_car_tracking )  )  ) ) {
    goal1_condition1.execute();
    goal1_condition1.raiseAlarm();
    return TRUE;
  }
  return FALSE;
}

void execute_goal1_condition1() {
  processactions(ac_bug_notification);
}

void alarm_goal1_condition1() {
  processalerts(al_bug_notification);
}

bool_t trig_goal2_condition1() {
  if (( stable_control_check() == stable_stable_control ) && ( (
       !( line_detection_check() == stable_line_detection )  ) &&
      ( ( front_car_tracking_check() ==
      disappeared_more_than_t1_front_car_tracking ) || (
      line_detection_check() ==
      no_detection_in_more_than_t6_line_detection ) || (
      straddling_car_tracking_check() ==
      straddling_more_than_t7_straddling_car_tracking )  )  ) ) {
    goal2_condition1.execute();
    goal2_condition1.raiseAlarm();
    return TRUE;
  }
```

```
  return FALSE;
}

void execute_goal2_condition1() {
  processactions(ac_emergency_operation_1);
}

void alarm_goal2_condition1() {
  processalerts(al_eop1);
}

bool_t trig_goal3_condition1() {
  if (front_car_distance_check() ==
      imminent_collision_distance_front_car_distance) {
    goal3_condition1.execute();
    goal3_condition1.raiseAlarm();
    return TRUE;
  }
  return FALSE;
}

void execute_goal3_condition1() {
  processactions(ac_report_longitudinal_delta_V);
}

void alarm_goal3_condition1() {
  processalerts(al_longitudinal_acceleration);
}

bool_t trig_goal3_condition2() {
  if (straddling_car_distance_check() ==
      imminent_collision_distance_straddling_car_distance) {
    goal3_condition2.execute();
    goal3_condition2.raiseAlarm();
    return TRUE;
  }
  return FALSE;
}

void execute_goal3_condition2() {
  processactions(ac_report_longitudinal_delta_V);
}

void alarm_goal3_condition2() {
  processalerts(al_longitudinal_acceleration);
}

bool_t trig_goal4_condition1() {
  if (( front_car_distance_check() ==
      imminent_collision_distance_front_car_distance ) || (
      straddling_car_distance_check() ==
```

```
     imminent_collision_distance_straddling_car_distance ) ) {
    goal4_condition1.execute();
    goal4_condition1.raiseAlarm();
    return TRUE;
  }
  return FALSE;
}

void execute_goal4_condition1() {
  processactions(ac_emergency_maneuver);
}

void alarm_goal4_condition1() {
  processalerts(al_longitudinal_acceleration);
}

bool_t trig_goal4_condition2() {
  if (front_car_tracking_check() ==
     not_confirmed_front_car_tracking) {
    goal4_condition2.execute();
    goal4_condition2.raiseAlarm();
    return TRUE;
  }
  return FALSE;
}

void execute_goal4_condition2() {
  processactions(ac_brake_acc);
}

void alarm_goal4_condition2() {
  processalerts(al_longitudinal_acceleration);
}

bool_t trig_goal4_condition3() {
  if (rear_car_distance_check() ==
     emergency_distance_rear_car_distance) {
    goal4_condition3.execute();
    goal4_condition3.raiseAlarm();
    return TRUE;
  }
  return FALSE;
}

void execute_goal4_condition3() {
  processactions(ac_no_action_needed);
}

void alarm_goal4_condition3() {
  processalerts(al_no_alert_needed);
}
```

```
bool_t trig_goal4_condition4() {
  if (( front_car_distance_check() ==
      strong_braking_distance_front_car_distance ) || (
      straddling_car_distance_check() ==
      strong_braking_distance_straddling_car_distance ) ) {
    goal4_condition4.execute();
    goal4_condition4.raiseAlarm();
    return TRUE;
  }
  return FALSE;
}

void execute_goal4_condition4() {
  processactions(ac_brake_strong);
}

void alarm_goal4_condition4() {
  processalerts(al_longitudinal_acceleration);
}

bool_t trig_goal4_condition5() {
  if (( front_car_distance_check() ==
      acc_distance_front_car_distance ) || (
      straddling_car_distance_check() ==
      acc_distance_straddling_car_distance ) ) {
    goal4_condition5.execute();
    goal4_condition5.raiseAlarm();
    return TRUE;
  }
  return FALSE;
}

void execute_goal4_condition5() {
  processactions(ac_brake_acc);
}

void alarm_goal4_condition5() {
  processalerts(al_longitudinal_acceleration);
}
```

Listing A.17: Safety_Checks.h

```
/** \file Safety_Checks.h
*
* \author
* \brief safety checker
*
*/
#ifndef Safety_Checks_H_
#define Safety_Checks_H_
```

```
typedef bool_t(isTriggFunct_t)(void);
typedef void(executeBehavior_t)(void);
typedef void(raiseAlarm_t)(void);

typedef struct Goal_t {
  isTriggFunct_t *isTriggered;
  executeBehavior_t *execute;
  raiseAlarm_t *raiseAlarm;
} Goal_t;

typedef enum {
  n_goal1_condition1,
  n_goal2_condition1,
  n_goal3_condition1,
  n_goal3_condition2,
  n_goal4_condition1,
  n_goal4_condition2,
  n_goal4_condition3,
  n_goal4_condition4,
  n_goal4_condition5,
  Goal_condition_size,
} goal_condition_t;

void init_goals();
void trig_goals();

Goal_t goal1_condition1;
bool_t trig_goal1_condition1();
void  execute_goal1_condition1();
void  alarm_goal1_condition1();

Goal_t goal2_condition1;
bool_t trig_goal2_condition1();
void  execute_goal2_condition1();
void  alarm_goal2_condition1();

Goal_t goal3_condition1;
bool_t trig_goal3_condition1();
void  execute_goal3_condition1();
void  alarm_goal3_condition1();

Goal_t goal3_condition2;
bool_t trig_goal3_condition2();
void  execute_goal3_condition2();
void  alarm_goal3_condition2();

Goal_t goal4_condition1;
bool_t trig_goal4_condition1();
void  execute_goal4_condition1();
void  alarm_goal4_condition1();
```

```
Goal_t goal4_condition2;
bool_t trig_goal4_condition2();
void  execute_goal4_condition2();
void  alarm_goal4_condition2();

Goal_t goal4_condition3;
bool_t trig_goal4_condition3();
void  execute_goal4_condition3();
void  alarm_goal4_condition3();

Goal_t goal4_condition4;
bool_t trig_goal4_condition4();
void  execute_goal4_condition4();
void  alarm_goal4_condition4();

Goal_t goal4_condition5;
bool_t trig_goal4_condition5();
void  execute_goal4_condition5();
void  alarm_goal4_condition5();


#endif
```

A.4 Java code auto-generation using EPSAAV generator

Listing A.18: Sat4jRules.java

```
package checksafety;
import java.util.Arrays;

import fr.kairos.timesquare.ccsl.sat.IBooleanSpecification;

public class Sat4jRules{

  static public void build_Equiv(IBooleanSpecification spec,
     String a, String b, String c) {
    spec.clause(a,b,c);
    spec.clause(b,a,c);
    spec.implies(a,c);
    spec.or(b+c,b,c);
    spec.forces(b+c);
  }

//  static public void build_Xor(IBooleanSpecification spec,
   String a, String b, String c) {
//    spec.clause(c,a,b);
//    spec.clause(b,a,c);
```

```
//    spec.clause(a,b,c);
//  }

  static public void build_goal1_cond1(IBooleanSpecification spec
     ) {
    //spec.not("goal1_cond1");
    spec.or("goal1_cond1", "goal1_cond1_1", "goal1_cond1_2");
    build_Equiv(spec,"goal1_cond1_1", "
       not_exist_front_car_distance",
          "goal1_cond1_1_1");
    spec.or("goal1_cond1_1_1", "not_exist_front_car_tracking",
        "disappeared_less_than_t1_front_car_tracking",
        "disappeared_more_than_t1_front_car_tracking");
    build_Equiv(spec,"goal1_cond1_2", "
       not_exist_straddling_car_distance",
          "not_exist_straddling_car_tracking");
  }

  static public void build_goal2_cond1(IBooleanSpecification spec
     ) {
    spec.and("goal2_cond1", "stable_stable_control", "
       goal2_cond1_1");
    spec.or("goal2_cond1_1",
        "disappeared_more_than_t1_front_car_tracking",
        "no_detection_in_more_than_t6_line_detection",
        "straddling_more_than_t7_straddling_car_tracking");
  }

  static public void build_goal3_cond1(IBooleanSpecification spec
     ) {
    spec.or("goal3_cond1", "
       imminent_collision_distance_front_car_distance");
  }

  static public void build_goal3_cond2(IBooleanSpecification spec
     ) {
    spec.or("goal3_cond2", "
       imminent_collision_distance_straddling_car_distance");
  }

  static public void build_goal4_cond1(IBooleanSpecification spec
     ) {
    spec.or("goal4_cond1", "
       imminent_collision_distance_front_car_distance", "
       imminent_collision_distance_straddling_car_distance");
  }

  static public void build_goal4_cond2(IBooleanSpecification spec
     ) {
    spec.or("goal4_cond2", "not_confirmed_front_car_tracking");
  }
```

```
  static public void build_goal4_cond3(IBooleanSpecification spec
      ) {
    spec.or("goal4_cond3", "emergency_distance_rear_car_distance
       ");
  }

  static public void build_goal4_cond4(IBooleanSpecification spec
      ) {
    spec.or("goal4_cond4", "
       strong_braking_distance_front_car_distance", "
       strong_braking_distance_straddling_car_distance");
  }

  static public void build_goal4_cond5(IBooleanSpecification spec
      ) {
    spec.or("goal4_cond5", "acc_distance_front_car_distance", "
       acc_distance_straddling_car_distance");
  }

}
```

Listing A.19: Sat4jRulesConsistency.java

```
package checksafety;
import fr.kairos.timesquare.ccsl.sat.IBooleanSpecification;

public class Sat4jRulesConsistency{

//Creating function *build_goal(x)_condition(y)_True*
//Creating function *build_goal(x)_condition(y)_False*
//Creating function *build_goal(x)_condition(y)_Alert*
//Creating function *build_goal(x)_condition(y)_Alert_True*
//Creating function *build_goal(x)_condition(y)_Alert_False*

  static public void build_goal1_cond1_True(IBooleanSpecification
      spec) {
    Sat4jRules.build_goal1_cond1(spec);
    spec.forces("goal1_cond1");
  }

  static public void build_goal1_cond1_False(
     IBooleanSpecification spec) {
    Sat4jRules.build_goal1_cond1(spec);
    spec.not("goal1_cond1");
  }

  static public void build_goal1_cond1_Alert(
     IBooleanSpecification spec) {
    Sat4jRules.build_goal1_cond1(spec);
    spec.and("goal1_cond1_Alert", "goal1_cond1","bug_notification
```

```
      ");
}

static public void build_goal1_cond1_Alert_True(
    IBooleanSpecification spec) {
  build_goal1_cond1_Alert(spec);
  spec.forces("goal1_cond1_Alert");
}

static public void build_goal1_cond1_Alert_False(
    IBooleanSpecification spec) {
  build_goal1_cond1_Alert(spec);
  spec.not("goal1_cond1_Alert");
}

static public void build_goal2_cond1_True(IBooleanSpecification
     spec) {
  Sat4jRules.build_goal2_cond1(spec);
  spec.forces("goal2_cond1");
}

static public void build_goal2_cond1_False(
    IBooleanSpecification spec) {
  Sat4jRules.build_goal2_cond1(spec);
  spec.not("goal2_cond1");
}

static public void build_goal2_cond1_Alert(
    IBooleanSpecification spec) {
  Sat4jRules.build_goal2_cond1(spec);
  spec.and("goal2_cond1_Alert", "goal2_cond1","eop1");
}

static public void build_goal2_cond1_Alert_True(
    IBooleanSpecification spec) {
  build_goal2_cond1_Alert(spec);
  spec.forces("goal2_cond1_Alert");
}

static public void build_goal2_cond1_Alert_False(
    IBooleanSpecification spec) {
  build_goal2_cond1_Alert(spec);
  spec.not("goal2_cond1_Alert");
}

static public void build_goal3_cond1_True(IBooleanSpecification
     spec) {
  Sat4jRules.build_goal3_cond1(spec);
  spec.forces("goal3_cond1");
}
```

```
static public void build_goal3_cond1_False(
    IBooleanSpecification spec) {
  Sat4jRules.build_goal3_cond1(spec);
  spec.not("goal3_cond1");
}

static public void build_goal3_cond1_Alert(
    IBooleanSpecification spec) {
  Sat4jRules.build_goal3_cond1(spec);
  spec.and("goal3_cond1_Alert", "goal3_cond1","
     longitudinal_acceleration");
}

static public void build_goal3_cond1_Alert_True(
    IBooleanSpecification spec) {
  build_goal3_cond1_Alert(spec);
  spec.forces("goal3_cond1_Alert");
}

static public void build_goal3_cond1_Alert_False(
    IBooleanSpecification spec) {
  build_goal3_cond1_Alert(spec);
  spec.not("goal3_cond1_Alert");
}
static public void build_goal3_cond2_True(IBooleanSpecification
     spec) {
  Sat4jRules.build_goal3_cond2(spec);
  spec.forces("goal3_cond2");
}

static public void build_goal3_cond2_False(
    IBooleanSpecification spec) {
  Sat4jRules.build_goal3_cond2(spec);
  spec.not("goal3_cond2");
}

static public void build_goal3_cond2_Alert(
    IBooleanSpecification spec) {
  Sat4jRules.build_goal3_cond2(spec);
  spec.and("goal3_cond2_Alert", "goal3_cond2","
     longitudinal_acceleration");
}

static public void build_goal3_cond2_Alert_True(
    IBooleanSpecification spec) {
  build_goal3_cond2_Alert(spec);
  spec.forces("goal3_cond2_Alert");
}

static public void build_goal3_cond2_Alert_False(
    IBooleanSpecification spec) {
```

```
  build_goal3_cond2_Alert(spec);
  spec.not("goal3_cond2_Alert");
}

static public void build_goal4_cond1_True(IBooleanSpecification
    spec) {
  Sat4jRules.build_goal4_cond1(spec);
  spec.forces("goal4_cond1");
}

static public void build_goal4_cond1_False(
   IBooleanSpecification spec) {
  Sat4jRules.build_goal4_cond1(spec);
  spec.not("goal4_cond1");
}

static public void build_goal4_cond1_Alert(
   IBooleanSpecification spec) {
  Sat4jRules.build_goal4_cond1(spec);
  spec.and("goal4_cond1_Alert", "goal4_cond1","
     longitudinal_acceleration");
}

static public void build_goal4_cond1_Alert_True(
   IBooleanSpecification spec) {
  build_goal4_cond1_Alert(spec);
  spec.forces("goal4_cond1_Alert");
}

static public void build_goal4_cond1_Alert_False(
   IBooleanSpecification spec) {
  build_goal4_cond1_Alert(spec);
  spec.not("goal4_cond1_Alert");
}
static public void build_goal4_cond2_True(IBooleanSpecification
    spec) {
  Sat4jRules.build_goal4_cond2(spec);
  spec.forces("goal4_cond2");
}

static public void build_goal4_cond2_False(
   IBooleanSpecification spec) {
  Sat4jRules.build_goal4_cond2(spec);
  spec.not("goal4_cond2");
}

static public void build_goal4_cond2_Alert(
   IBooleanSpecification spec) {
  Sat4jRules.build_goal4_cond2(spec);
  spec.and("goal4_cond2_Alert", "goal4_cond2","
     longitudinal_acceleration");
```

```
}

static public void build_goal4_cond2_Alert_True(
    IBooleanSpecification spec) {
  build_goal4_cond2_Alert(spec);
  spec.forces("goal4_cond2_Alert");
}

static public void build_goal4_cond2_Alert_False(
    IBooleanSpecification spec) {
  build_goal4_cond2_Alert(spec);
  spec.not("goal4_cond2_Alert");
}
static public void build_goal4_cond3_True(IBooleanSpecification
     spec) {
  Sat4jRules.build_goal4_cond3(spec);
  spec.forces("goal4_cond3");
}

static public void build_goal4_cond3_False(
    IBooleanSpecification spec) {
  Sat4jRules.build_goal4_cond3(spec);
  spec.not("goal4_cond3");
}

static public void build_goal4_cond3_Alert(
    IBooleanSpecification spec) {
  Sat4jRules.build_goal4_cond3(spec);
  spec.and("goal4_cond3_Alert", "goal4_cond3","no_alert_needed
      ");
}

static public void build_goal4_cond3_Alert_True(
    IBooleanSpecification spec) {
  build_goal4_cond3_Alert(spec);
  spec.forces("goal4_cond3_Alert");
}

static public void build_goal4_cond3_Alert_False(
    IBooleanSpecification spec) {
  build_goal4_cond3_Alert(spec);
  spec.not("goal4_cond3_Alert");
}
static public void build_goal4_cond4_True(IBooleanSpecification
     spec) {
  Sat4jRules.build_goal4_cond4(spec);
  spec.forces("goal4_cond4");
}

static public void build_goal4_cond4_False(
    IBooleanSpecification spec) {
```

```
  Sat4jRules.build_goal4_cond4(spec);
  spec.not("goal4_cond4");
}

static public void build_goal4_cond4_Alert(
    IBooleanSpecification spec) {
  Sat4jRules.build_goal4_cond4(spec);
  spec.and("goal4_cond4_Alert", "goal4_cond4","
      longitudinal_acceleration");
}

static public void build_goal4_cond4_Alert_True(
    IBooleanSpecification spec) {
  build_goal4_cond4_Alert(spec);
  spec.forces("goal4_cond4_Alert");
}

static public void build_goal4_cond4_Alert_False(
    IBooleanSpecification spec) {
  build_goal4_cond4_Alert(spec);
  spec.not("goal4_cond4_Alert");
}
static public void build_goal4_cond5_True(IBooleanSpecification
     spec) {
  Sat4jRules.build_goal4_cond5(spec);
  spec.forces("goal4_cond5");
}

static public void build_goal4_cond5_False(
    IBooleanSpecification spec) {
  Sat4jRules.build_goal4_cond5(spec);
  spec.not("goal4_cond5");
}

static public void build_goal4_cond5_Alert(
    IBooleanSpecification spec) {
  Sat4jRules.build_goal4_cond5(spec);
  spec.and("goal4_cond5_Alert", "goal4_cond5","
      longitudinal_acceleration");
}

static public void build_goal4_cond5_Alert_True(
    IBooleanSpecification spec) {
  build_goal4_cond5_Alert(spec);
  spec.forces("goal4_cond5_Alert");
}

static public void build_goal4_cond5_Alert_False(
    IBooleanSpecification spec) {
  build_goal4_cond5_Alert(spec);
  spec.not("goal4_cond5_Alert");
```

```
  }

//test priority rules by applying : test(cond1 and !cond2) ou
    test(!cond1 and cond2)


//  static public void testPrioritygoal3_cond2_over_goal3_cond1(
    IBooleanSpecification spec) {
//    build_goal3_cond2_True(spec);
//    build_goal3_cond1_False(spec);
//  }
//
//  static public void testPrioritygoal3_cond1_over_goal3_cond2(
    IBooleanSpecification spec) {
//    build_goal3_cond2_False(spec);
//    build_goal3_cond1_True(spec);
//  }
//
//  static public void
    testPrioritygoal3_cond2_over_goal3_cond1_Alerts(
    IBooleanSpecification spec) {
//    build_goal3_cond2_Alert_True(spec);
//    build_goal3_cond1_Alert_False(spec);
//  }
//
//  static public void
    testPrioritygoal3_cond1_over_goal3_cond2_Alerts(
    IBooleanSpecification spec) {
//    build_goal3_cond2_Alert_False(spec);
//    build_goal3_cond1_Alert_True(spec);
//  }
//

  static public void testPrioritygoal4_cond2_over_goal4_cond1(
     IBooleanSpecification spec) {
    build_goal4_cond2_True(spec);
    build_goal4_cond1_False(spec);
  }

//  static public void testPrioritygoal4_cond1_over_goal4_cond2(
    IBooleanSpecification spec) {
//    build_goal4_cond2_False(spec);
//    build_goal4_cond1_True(spec);
//  }

  static public void
     testPrioritygoal4_cond2_over_goal4_cond1_Alerts(
     IBooleanSpecification spec) {
    build_goal4_cond2_Alert_True(spec);
    build_goal4_cond1_Alert_False(spec);
  }
```

```
//  static public void
    testPrioritygoal4_cond1_over_goal4_cond2_Alerts(
    IBooleanSpecification spec) {
//    build_goal4_cond2_Alert_False(spec);
//    build_goal4_cond1_Alert_True(spec);
//  }

  static public void testPrioritygoal4_cond3_over_goal4_cond2(
      IBooleanSpecification spec) {
    build_goal4_cond3_True(spec);
    build_goal4_cond2_False(spec);
  }

//  static public void testPrioritygoal4_cond2_over_goal4_cond3(
    IBooleanSpecification spec) {
//    build_goal4_cond3_False(spec);
//    build_goal4_cond2_True(spec);
//  }

  static public void
      testPrioritygoal4_cond3_over_goal4_cond2_Alerts(
      IBooleanSpecification spec) {
    build_goal4_cond3_Alert_True(spec);
    build_goal4_cond2_Alert_False(spec);
  }

//  static public void
    testPrioritygoal4_cond2_over_goal4_cond3_Alerts(
    IBooleanSpecification spec) {
//    build_goal4_cond3_Alert_False(spec);
//    build_goal4_cond2_Alert_True(spec);
//  }

  static public void testPrioritygoal4_cond4_over_goal4_cond3(
      IBooleanSpecification spec) {
    build_goal4_cond4_True(spec);
    build_goal4_cond3_False(spec);
  }

//  static public void testPrioritygoal4_cond3_over_goal4_cond4(
    IBooleanSpecification spec) {
//    build_goal4_cond4_False(spec);
//    build_goal4_cond3_True(spec);
//  }

  static public void
      testPrioritygoal4_cond4_over_goal4_cond3_Alerts(
      IBooleanSpecification spec) {
    build_goal4_cond4_Alert_True(spec);
    build_goal4_cond3_Alert_False(spec);
```

```
  }

//  static public void
    testPrioritygoal4_cond3_over_goal4_cond4_Alerts(
    IBooleanSpecification spec) {
//    build_goal4_cond4_Alert_False(spec);
//    build_goal4_cond3_Alert_True(spec);
//  }

  static public void testPrioritygoal4_cond5_over_goal4_cond4(
      IBooleanSpecification spec) {
    build_goal4_cond5_True(spec);
    build_goal4_cond4_False(spec);
  }

//  static public void testPrioritygoal4_cond4_over_goal4_cond5(
    IBooleanSpecification spec) {
//    build_goal4_cond5_False(spec);
//    build_goal4_cond4_True(spec);
//  }

  static public void
     testPrioritygoal4_cond5_over_goal4_cond4_Alerts(
     IBooleanSpecification spec) {
    build_goal4_cond5_Alert_True(spec);
    build_goal4_cond4_Alert_False(spec);
  }

//  static public void
    testPrioritygoal4_cond4_over_goal4_cond5_Alerts(
    IBooleanSpecification spec) {
//    build_goal4_cond5_Alert_False(spec);
//    build_goal4_cond4_Alert_True(spec);
//  }


//  static public void testPrioritygoal1_cond1_over_goal2_cond1(
    IBooleanSpecification spec) {
//    build_goal1_cond1_True(spec);
//    build_goal2_cond1_False(spec);
//  }

  static public void testPrioritygoal2_cond1_over_goal1_cond1(
      IBooleanSpecification spec) {
    build_goal1_cond1_True(spec);
    build_goal2_cond1_True(spec);
  }

//  static public void
    testPrioritygoal1_cond1_over_goal2_cond1_Alerts(
    IBooleanSpecification spec) {
```

```
//    build_goal1_cond1_Alert_True(spec);
//    build_goal2_cond1_Alert_False(spec);
//  }

  static public void
     testPrioritygoal2_cond1_over_goal1_cond1_Alerts(
     IBooleanSpecification spec) {
    build_goal1_cond1_Alert_True(spec);
    build_goal2_cond1_Alert_True(spec);
  }

//  static public void testPrioritygoal2_cond1_over_goal3_cond1(
    IBooleanSpecification spec) {
//    build_goal2_cond1_True(spec);
//    build_goal3_cond1_False(spec);
//  }
//
//  static public void testPrioritygoal3_cond1_over_goal2_cond1(
    IBooleanSpecification spec) {
//    build_goal2_cond1_False(spec);
//    build_goal3_cond1_True(spec);
//  }
//
//  static public void
    testPrioritygoal2_cond1_over_goal3_cond1_Alerts(
    IBooleanSpecification spec) {
//    build_goal2_cond1_Alert_True(spec);
//    build_goal3_cond1_Alert_False(spec);
//  }
//
//  static public void
    testPrioritygoal3_cond1_over_goal2_cond1_Alerts(
    IBooleanSpecification spec) {
//    build_goal2_cond1_Alert_False(spec);
//    build_goal3_cond1_Alert_True(spec);
//  }

//  static public void testPrioritygoal3_cond2_over_goal4_cond1(
    IBooleanSpecification spec) {
//    build_goal3_cond2_True(spec);
//    build_goal4_cond1_False(spec);
//  }
//
//  static public void testPrioritygoal4_cond1_over_goal3_cond2(
    IBooleanSpecification spec) {
//    build_goal3_cond2_False(spec);
//    build_goal4_cond1_True(spec);
//  }
//
//  static public void
    testPrioritygoal3_cond2_over_goal4_cond1_Alerts(
```

```
    IBooleanSpecification spec) {
//    build_goal3_cond2_Alert_True(spec);
//    build_goal4_cond1_Alert_False(spec);
//  }
//
//  static public void
    testPrioritygoal4_cond1_over_goal3_cond2_Alerts(
    IBooleanSpecification spec) {
//    build_goal3_cond2_Alert_False(spec);
//    build_goal4_cond1_Alert_True(spec);
//  }

//test parallel rules by applying : test(!goalX_cond1 and !
    goalX_cond2)


  static public void testParallelgoal3_cond1_goal3_cond2(
      IBooleanSpecification spec) {
    build_goal3_cond2_True(spec);
    build_goal3_cond1_True(spec);
  }

  static public void testParallelgoal3_cond1_goal3_cond2_Alerts(
      IBooleanSpecification spec) {
    build_goal3_cond2_Alert_True(spec);
    build_goal3_cond1_Alert_True(spec);
  }


//  static public void testParallelgoal4_cond1_goal4_cond2(
    IBooleanSpecification spec) {
//    build_goal4_cond2_True(spec);
//    build_goal4_cond1_True(spec);
//  }
//
//  static public void testParallelgoal4_cond1_goal4_cond2_Alerts
    (IBooleanSpecification spec) {
//    build_goal4_cond2_Alert_True(spec);
//    build_goal4_cond1_Alert_True(spec);
//  }
//
//  static public void testParallelgoal4_cond2_goal4_cond3(
    IBooleanSpecification spec) {
//    build_goal4_cond3_True(spec);
//    build_goal4_cond2_True(spec);
//  }
//
//  static public void testParallelgoal4_cond2_goal4_cond3_Alerts
    (IBooleanSpecification spec) {
//    build_goal4_cond3_Alert_True(spec);
//    build_goal4_cond2_Alert_True(spec);
```

```
//  }
//
//  static public void testParallelgoal4_cond3_goal4_cond4(
    IBooleanSpecification spec) {
//    build_goal4_cond4_True(spec);
//    build_goal4_cond3_True(spec);
//  }
//
//  static public void testParallelgoal4_cond3_goal4_cond4_Alerts
    (IBooleanSpecification spec) {
//    build_goal4_cond4_Alert_True(spec);
//    build_goal4_cond3_Alert_True(spec);
//  }
//
//  static public void testParallelgoal4_cond4_goal4_cond5(
    IBooleanSpecification spec) {
//    build_goal4_cond5_True(spec);
//    build_goal4_cond4_True(spec);
//  }
//
//  static public void testParallelgoal4_cond4_goal4_cond5_Alerts
    (IBooleanSpecification spec) {
//    build_goal4_cond5_Alert_True(spec);
//    build_goal4_cond4_Alert_True(spec);
//  }


//  static public void testParallelgoal1_cond1_goal2_cond1(
    IBooleanSpecification spec) {
//    build_goal1_cond1_True(spec);
//    build_goal2_cond1_True(spec);
//  }
//
//  static public void testParallelgoal1_cond1_goal2_cond1_Alerts
    (IBooleanSpecification spec) {
//    build_goal1_cond1_Alert_True(spec);
//    build_goal2_cond1_Alert_True(spec);
//  }

  static public void testParallelgoal2_cond1_goal3_cond1(
      IBooleanSpecification spec) {
    build_goal2_cond1_True(spec);
    build_goal3_cond1_True(spec);
  }

  static public void testParallelgoal2_cond1_goal3_cond1_Alerts(
      IBooleanSpecification spec) {
    build_goal2_cond1_Alert_True(spec);
    build_goal3_cond1_Alert_True(spec);
  }
```

```
  static public void testParallelgoal3_cond2_goal4_cond1(
      IBooleanSpecification spec) {
    build_goal3_cond2_True(spec);
    build_goal4_cond1_True(spec);
  }

  static public void testParallelgoal3_cond2_goal4_cond1_Alerts(
      IBooleanSpecification spec) {
    build_goal3_cond2_Alert_True(spec);
    build_goal4_cond1_Alert_True(spec);
  }

}
```

Listing A.20: Sat4jSystemConsistency.java

```
package checksafety;
import fr.kairos.timesquare.ccsl.sat.IBooleanSpecification;

public class Sat4jSystemConsistency{

  static public void system_solution(IBooleanSpecification spec){
    coherence_all_properties(spec);
    coherence_all_goals_behaviors(spec);
    spec.and("system_solution", "coherence_all_properties", "
       coherence_all_goals_behaviors");
  }

  static public void coherence_all_goals_behaviors(
      IBooleanSpecification spec){
    coherence_goal1(spec);
    coherence_goal2(spec);
    coherence_goal3(spec);
    coherence_goal4(spec);
    spec.or("coherence_all_goals_behaviors", "coherence_goal1","
       coherence_goal2",
        "coherence_goal3","coherence_goal4");
  }

  static public void coherence_all_goals_behaviors_true(
      IBooleanSpecification spec){
    coherence_all_goals_behaviors(spec);
    spec.forces("coherence_all_goals_behaviors");
  }

  static public void coherence_goal1(IBooleanSpecification spec){
    //Sat4jRulesConsistency.build_goal1_cond1_Alert(spec);
    Sat4jRules.build_goal1_cond1(spec);
    spec.or("coherence_goal1", "goal1_cond1");
  }
```

```
static public void coherence_goal1_FALSE(IBooleanSpecification
    spec){
  //Sat4jRulesConsistency.build_goal1_cond1_Alert_False(spec);
  Sat4jRulesConsistency.build_goal1_cond1_False(spec);
}

static public void coherence_goal2(IBooleanSpecification spec){
  coherence_goal1_FALSE(spec);
  //Sat4jRulesConsistency.build_goal2_cond1_Alert(spec);
  Sat4jRules.build_goal2_cond1(spec);
  spec.or("coherence_goal2", "goal2_cond1");
}

static public void coherence_goal3(IBooleanSpecification spec){
  coherence_goal1_FALSE(spec);
  //Sat4jRulesConsistency.build_goal3_cond1_Alert(spec);
  //Sat4jRulesConsistency.build_goal3_cond2_Alert(spec);
  Sat4jRules.build_goal3_cond1(spec);
  Sat4jRules.build_goal3_cond2(spec);
  spec.or("coherence_goal3", "goal3_cond1","goal3_cond2");

}

static public void coherence_goal4(IBooleanSpecification spec){
  coherence_goal1_FALSE(spec);
  coherence_goal4_cond1(spec);
  coherence_goal4_cond2(spec);
  coherence_goal4_cond3(spec);
  coherence_goal4_cond4(spec);
  coherence_goal4_cond5(spec);
  spec.or("coherence_goal4", "goal4_cond1","goal4_cond2","
     goal4_cond3","goal4_cond4","goal4_cond5");
}

static public void coherence_goal4_cond1(IBooleanSpecification
    spec){
  //Sat4jRulesConsistency.build_goal4_cond1_Alert(spec);
  Sat4jRules.build_goal4_cond1(spec);
}


static public void coherence_goal4_cond1_FALSE(
    IBooleanSpecification spec){
  //Sat4jRulesConsistency.build_goal4_cond1_Alert_False(spec);
  Sat4jRulesConsistency.build_goal4_cond1_False(spec);
}

static public void coherence_goal4_cond2(IBooleanSpecification
    spec){
  coherence_goal4_cond1_FALSE(spec);
  //Sat4jRulesConsistency.build_goal4_cond2_Alert(spec);
```

```
  Sat4jRules.build_goal4_cond2(spec);

}

static public void coherence_goal4_cond2_FALSE(
    IBooleanSpecification spec){
  //Sat4jRulesConsistency.build_goal4_cond2_Alert_False(spec);
  Sat4jRulesConsistency.build_goal4_cond2_False(spec);
}

static public void coherence_goal4_cond3(IBooleanSpecification
    spec){
  coherence_goal4_cond2_FALSE(spec);
  //Sat4jRulesConsistency.build_goal4_cond3_Alert(spec);
  Sat4jRules.build_goal4_cond3(spec);
}

static public void coherence_goal4_cond3_FALSE(
    IBooleanSpecification spec){
  //Sat4jRulesConsistency.build_goal4_cond3_Alert_False(spec);
  Sat4jRulesConsistency.build_goal4_cond3_False(spec);
}

static public void coherence_goal4_cond4(IBooleanSpecification
    spec){
  coherence_goal4_cond3_FALSE(spec);
  //Sat4jRulesConsistency.build_goal4_cond4_Alert(spec);
  Sat4jRules.build_goal4_cond4(spec);
}

static public void coherence_goal4_cond4_FALSE(
    IBooleanSpecification spec){
  //Sat4jRulesConsistency.build_goal4_cond4_Alert_False(spec);
  Sat4jRulesConsistency.build_goal4_cond4_False(spec);
}

static public void coherence_goal4_cond5(IBooleanSpecification
    spec){
  coherence_goal4_cond4_FALSE(spec);
  //Sat4jRulesConsistency.build_goal4_cond5_Alert(spec);
  Sat4jRules.build_goal4_cond5(spec);
}


static public void coherence_all_properties(
    IBooleanSpecification spec){
  coherence_traffic_jam(spec);
  coherence_front_car_distance(spec);
  coherence_front_car_tracking(spec);
  coherence_line_detection(spec);
  coherence_straddling_car_distance(spec);
```

```
  coherence_straddling_car_tracking(spec);
  coherence_stable_control(spec);
  coherence_rear_car_distance(spec);
  spec.or("coherence_all_properties",  "coherence_traffic_jam",
      "coherence_front_car_distance", "
         coherence_front_car_tracking",
      "coherence_line_detection", "
         coherence_straddling_car_distance",
      "coherence_straddling_car_tracking", "
         coherence_stable_control",
      "coherence_rear_car_distance");
}

static public void coherence_traffic_jam(IBooleanSpecification
    spec){
  spec.clause("yes_traffic_jam", "no_traffic_jam","
     coherence_traffic_jam");
  spec.clause("no_traffic_jam", "yes_traffic_jam","
     coherence_traffic_jam");
  spec.clause("coherence_traffic_jam", "yes_traffic_jam", "
     no_traffic_jam");
  spec.forbids("yes_traffic_jam","no_traffic_jam");
  spec.forbids("no_traffic_jam","yes_traffic_jam");
  spec.not("coherence_traffic_jam");
}


static public void coherence_front_car_distance(
    IBooleanSpecification spec){
  spec.clause("not_exist_front_car_distance", "
     safe_distance_front_car_distance","
     acc_distance_front_car_distance","
     strong_braking_distance_front_car_distance","
     imminent_collision_distance_front_car_distance","
     coherence_front_car_distance");
  spec.clause("safe_distance_front_car_distance", "
     not_exist_front_car_distance","
     acc_distance_front_car_distance","
     strong_braking_distance_front_car_distance","
     imminent_collision_distance_front_car_distance","
     coherence_front_car_distance");
  spec.clause("acc_distance_front_car_distance", "
     not_exist_front_car_distance","
     safe_distance_front_car_distance","
     strong_braking_distance_front_car_distance","
     imminent_collision_distance_front_car_distance","
     coherence_front_car_distance");
  spec.clause("strong_braking_distance_front_car_distance", "
     not_exist_front_car_distance","
     safe_distance_front_car_distance","
     acc_distance_front_car_distance","
```

```
    imminent_collision_distance_front_car_distance","
    coherence_front_car_distance");
spec.clause("imminent_collision_distance_front_car_distance",
     "not_exist_front_car_distance","
    safe_distance_front_car_distance","
    acc_distance_front_car_distance","
    strong_braking_distance_front_car_distance","
    coherence_front_car_distance");
spec.clause("coherence_front_car_distance", "
    not_exist_front_car_distance", "
    safe_distance_front_car_distance", "
    acc_distance_front_car_distance", "
    strong_braking_distance_front_car_distance", "
    imminent_collision_distance_front_car_distance");
spec.forbids("not_exist_front_car_distance","
    safe_distance_front_car_distance");
spec.forbids("not_exist_front_car_distance","
    acc_distance_front_car_distance");
spec.forbids("not_exist_front_car_distance","
    strong_braking_distance_front_car_distance");
spec.forbids("not_exist_front_car_distance","
    imminent_collision_distance_front_car_distance");
spec.forbids("safe_distance_front_car_distance","
    not_exist_front_car_distance");
spec.forbids("safe_distance_front_car_distance","
    acc_distance_front_car_distance");
spec.forbids("safe_distance_front_car_distance","
    strong_braking_distance_front_car_distance");
spec.forbids("safe_distance_front_car_distance","
    imminent_collision_distance_front_car_distance");
spec.forbids("acc_distance_front_car_distance","
    not_exist_front_car_distance");
spec.forbids("acc_distance_front_car_distance","
    safe_distance_front_car_distance");
spec.forbids("acc_distance_front_car_distance","
    strong_braking_distance_front_car_distance");
spec.forbids("acc_distance_front_car_distance","
    imminent_collision_distance_front_car_distance");
spec.forbids("strong_braking_distance_front_car_distance","
    not_exist_front_car_distance");
spec.forbids("strong_braking_distance_front_car_distance","
    safe_distance_front_car_distance");
spec.forbids("strong_braking_distance_front_car_distance","
    acc_distance_front_car_distance");
spec.forbids("strong_braking_distance_front_car_distance","
    imminent_collision_distance_front_car_distance");
spec.forbids("imminent_collision_distance_front_car_distance
    ","not_exist_front_car_distance");
spec.forbids("imminent_collision_distance_front_car_distance
    ","safe_distance_front_car_distance");
spec.forbids("imminent_collision_distance_front_car_distance
```

```
      ","acc_distance_front_car_distance");
  spec.forbids("imminent_collision_distance_front_car_distance
      ","strong_braking_distance_front_car_distance");
  spec.not("coherence_front_car_distance");
}


static public void coherence_front_car_tracking(
    IBooleanSpecification spec){
  spec.clause("not_confirmed_front_car_tracking", "
      not_exist_front_car_tracking","
      disappeared_less_than_t1_front_car_tracking","
      disappeared_more_than_t1_front_car_tracking","
      stable_tracking_front_car_tracking","
      coherence_front_car_tracking");
  spec.clause("not_exist_front_car_tracking", "
      not_confirmed_front_car_tracking","
      disappeared_less_than_t1_front_car_tracking","
      disappeared_more_than_t1_front_car_tracking","
      stable_tracking_front_car_tracking","
      coherence_front_car_tracking");
  spec.clause("disappeared_less_than_t1_front_car_tracking", "
      not_confirmed_front_car_tracking","
      not_exist_front_car_tracking","
      disappeared_more_than_t1_front_car_tracking","
      stable_tracking_front_car_tracking","
      coherence_front_car_tracking");
  spec.clause("disappeared_more_than_t1_front_car_tracking", "
      not_confirmed_front_car_tracking","
      not_exist_front_car_tracking","
      disappeared_less_than_t1_front_car_tracking","
      stable_tracking_front_car_tracking","
      coherence_front_car_tracking");
  spec.clause("stable_tracking_front_car_tracking", "
      not_confirmed_front_car_tracking","
      not_exist_front_car_tracking","
      disappeared_less_than_t1_front_car_tracking","
      disappeared_more_than_t1_front_car_tracking","
      coherence_front_car_tracking");
  spec.clause("coherence_front_car_tracking", "
      not_confirmed_front_car_tracking", "
      not_exist_front_car_tracking", "
      disappeared_less_than_t1_front_car_tracking", "
      disappeared_more_than_t1_front_car_tracking", "
      stable_tracking_front_car_tracking");
  spec.forbids("not_confirmed_front_car_tracking","
      not_exist_front_car_tracking");
  spec.forbids("not_confirmed_front_car_tracking","
      disappeared_less_than_t1_front_car_tracking");
  spec.forbids("not_confirmed_front_car_tracking","
      disappeared_more_than_t1_front_car_tracking");
```

```
  spec.forbids("not_confirmed_front_car_tracking","
     stable_tracking_front_car_tracking");
  spec.forbids("not_exist_front_car_tracking","
     not_confirmed_front_car_tracking");
  spec.forbids("not_exist_front_car_tracking","
     disappeared_less_than_t1_front_car_tracking");
  spec.forbids("not_exist_front_car_tracking","
     disappeared_more_than_t1_front_car_tracking");
  spec.forbids("not_exist_front_car_tracking","
     stable_tracking_front_car_tracking");
  spec.forbids("disappeared_less_than_t1_front_car_tracking","
     not_confirmed_front_car_tracking");
  spec.forbids("disappeared_less_than_t1_front_car_tracking","
     not_exist_front_car_tracking");
  spec.forbids("disappeared_less_than_t1_front_car_tracking","
     disappeared_more_than_t1_front_car_tracking");
  spec.forbids("disappeared_less_than_t1_front_car_tracking","
     stable_tracking_front_car_tracking");
  spec.forbids("disappeared_more_than_t1_front_car_tracking","
     not_confirmed_front_car_tracking");
  spec.forbids("disappeared_more_than_t1_front_car_tracking","
     not_exist_front_car_tracking");
  spec.forbids("disappeared_more_than_t1_front_car_tracking","
     disappeared_less_than_t1_front_car_tracking");
  spec.forbids("disappeared_more_than_t1_front_car_tracking","
     stable_tracking_front_car_tracking");
  spec.forbids("stable_tracking_front_car_tracking","
     not_confirmed_front_car_tracking");
  spec.forbids("stable_tracking_front_car_tracking","
     not_exist_front_car_tracking");
  spec.forbids("stable_tracking_front_car_tracking","
     disappeared_less_than_t1_front_car_tracking");
  spec.forbids("stable_tracking_front_car_tracking","
     disappeared_more_than_t1_front_car_tracking");
  spec.not("coherence_front_car_tracking");
}


static public void coherence_line_detection(
    IBooleanSpecification spec){
  spec.clause("stable_line_detection", "
     no_detection_in_less_than_t6_line_detection","
     no_detection_in_more_than_t6_line_detection","
     coherence_line_detection");
  spec.clause("no_detection_in_less_than_t6_line_detection", "
     stable_line_detection","
     no_detection_in_more_than_t6_line_detection","
     coherence_line_detection");
  spec.clause("no_detection_in_more_than_t6_line_detection", "
     stable_line_detection","
     no_detection_in_less_than_t6_line_detection","
```

```
      coherence_line_detection");
  spec.clause("coherence_line_detection", "
      stable_line_detection", "
      no_detection_in_less_than_t6_line_detection", "
      no_detection_in_more_than_t6_line_detection");
  spec.forbids("stable_line_detection","
      no_detection_in_less_than_t6_line_detection");
  spec.forbids("stable_line_detection","
      no_detection_in_more_than_t6_line_detection");
  spec.forbids("no_detection_in_less_than_t6_line_detection","
      stable_line_detection");
  spec.forbids("no_detection_in_less_than_t6_line_detection","
      no_detection_in_more_than_t6_line_detection");
  spec.forbids("no_detection_in_more_than_t6_line_detection","
      stable_line_detection");
  spec.forbids("no_detection_in_more_than_t6_line_detection","
      no_detection_in_less_than_t6_line_detection");
  spec.not("coherence_line_detection");
}


static public void coherence_straddling_car_distance(
    IBooleanSpecification spec){
  spec.clause("
      imminent_collision_distance_straddling_car_distance", "
      safe_distance_straddling_car_distance","
      acc_distance_straddling_car_distance","
      strong_braking_distance_straddling_car_distance","
      not_exist_straddling_car_distance","
      coherence_straddling_car_distance");
  spec.clause("safe_distance_straddling_car_distance", "
      imminent_collision_distance_straddling_car_distance","
      acc_distance_straddling_car_distance","
      strong_braking_distance_straddling_car_distance","
      not_exist_straddling_car_distance","
      coherence_straddling_car_distance");
  spec.clause("acc_distance_straddling_car_distance", "
      imminent_collision_distance_straddling_car_distance","
      safe_distance_straddling_car_distance","
      strong_braking_distance_straddling_car_distance","
      not_exist_straddling_car_distance","
      coherence_straddling_car_distance");
  spec.clause("strong_braking_distance_straddling_car_distance
      ", "imminent_collision_distance_straddling_car_distance","
      safe_distance_straddling_car_distance","
      acc_distance_straddling_car_distance","
      not_exist_straddling_car_distance","
      coherence_straddling_car_distance");
  spec.clause("not_exist_straddling_car_distance", "
      imminent_collision_distance_straddling_car_distance","
      safe_distance_straddling_car_distance","
```

```
    acc_distance_straddling_car_distance","
    strong_braking_distance_straddling_car_distance","
    coherence_straddling_car_distance");
spec.clause("coherence_straddling_car_distance", "
    imminent_collision_distance_straddling_car_distance", "
    safe_distance_straddling_car_distance", "
    acc_distance_straddling_car_distance", "
    strong_braking_distance_straddling_car_distance", "
    not_exist_straddling_car_distance");
spec.forbids("
    imminent_collision_distance_straddling_car_distance","
    safe_distance_straddling_car_distance");
spec.forbids("
    imminent_collision_distance_straddling_car_distance","
    acc_distance_straddling_car_distance");
spec.forbids("
    imminent_collision_distance_straddling_car_distance","
    strong_braking_distance_straddling_car_distance");
spec.forbids("
    imminent_collision_distance_straddling_car_distance","
    not_exist_straddling_car_distance");
spec.forbids("safe_distance_straddling_car_distance","
    imminent_collision_distance_straddling_car_distance");
spec.forbids("safe_distance_straddling_car_distance","
    acc_distance_straddling_car_distance");
spec.forbids("safe_distance_straddling_car_distance","
    strong_braking_distance_straddling_car_distance");
spec.forbids("safe_distance_straddling_car_distance","
    not_exist_straddling_car_distance");
spec.forbids("acc_distance_straddling_car_distance","
    imminent_collision_distance_straddling_car_distance");
spec.forbids("acc_distance_straddling_car_distance","
    safe_distance_straddling_car_distance");
spec.forbids("acc_distance_straddling_car_distance","
    strong_braking_distance_straddling_car_distance");
spec.forbids("acc_distance_straddling_car_distance","
    not_exist_straddling_car_distance");
spec.forbids("strong_braking_distance_straddling_car_distance
    ","imminent_collision_distance_straddling_car_distance");
spec.forbids("strong_braking_distance_straddling_car_distance
    ","safe_distance_straddling_car_distance");
spec.forbids("strong_braking_distance_straddling_car_distance
    ","acc_distance_straddling_car_distance");
spec.forbids("strong_braking_distance_straddling_car_distance
    ","not_exist_straddling_car_distance");
spec.forbids("not_exist_straddling_car_distance","
    imminent_collision_distance_straddling_car_distance");
spec.forbids("not_exist_straddling_car_distance","
    safe_distance_straddling_car_distance");
spec.forbids("not_exist_straddling_car_distance","
    acc_distance_straddling_car_distance");
```

```
  spec.forbids("not_exist_straddling_car_distance","
     strong_braking_distance_straddling_car_distance");
  spec.not("coherence_straddling_car_distance");
}


static public void coherence_straddling_car_tracking(
    IBooleanSpecification spec){
  spec.clause("not_exist_straddling_car_tracking", "
     straddling_less_than_t7_straddling_car_tracking","
     straddling_more_than_t7_straddling_car_tracking","
     coherence_straddling_car_tracking");
  spec.clause("straddling_less_than_t7_straddling_car_tracking
     ", "not_exist_straddling_car_tracking","
     straddling_more_than_t7_straddling_car_tracking","
     coherence_straddling_car_tracking");
  spec.clause("straddling_more_than_t7_straddling_car_tracking
     ", "not_exist_straddling_car_tracking","
     straddling_less_than_t7_straddling_car_tracking","
     coherence_straddling_car_tracking");
  spec.clause("coherence_straddling_car_tracking", "
     not_exist_straddling_car_tracking", "
     straddling_less_than_t7_straddling_car_tracking", "
     straddling_more_than_t7_straddling_car_tracking");
  spec.forbids("not_exist_straddling_car_tracking","
     straddling_less_than_t7_straddling_car_tracking");
  spec.forbids("not_exist_straddling_car_tracking","
     straddling_more_than_t7_straddling_car_tracking");
  spec.forbids("straddling_less_than_t7_straddling_car_tracking
     ","not_exist_straddling_car_tracking");
  spec.forbids("straddling_less_than_t7_straddling_car_tracking
     ","straddling_more_than_t7_straddling_car_tracking");
  spec.forbids("straddling_more_than_t7_straddling_car_tracking
     ","not_exist_straddling_car_tracking");
  spec.forbids("straddling_more_than_t7_straddling_car_tracking
     ","straddling_less_than_t7_straddling_car_tracking");
  spec.not("coherence_straddling_car_tracking");
}


static public void coherence_stable_control(
    IBooleanSpecification spec){
  spec.clause("stable_stable_control", "
     not_stable_stable_control","coherence_stable_control");
  spec.clause("not_stable_stable_control", "
     stable_stable_control","coherence_stable_control");
  spec.clause("coherence_stable_control", "
     stable_stable_control", "not_stable_stable_control");
  spec.forbids("stable_stable_control","
     not_stable_stable_control");
  spec.forbids("not_stable_stable_control","
```

```
      stable_stable_control");
    spec.not("coherence_stable_control");
  }


  static public void coherence_rear_car_distance(
     IBooleanSpecification spec){
    spec.clause("not_exist_rear_car_distance", "
       safe_distance_rear_car_distance","
       emergency_distance_rear_car_distance","
       coherence_rear_car_distance");
    spec.clause("safe_distance_rear_car_distance", "
       not_exist_rear_car_distance","
       emergency_distance_rear_car_distance","
       coherence_rear_car_distance");
    spec.clause("emergency_distance_rear_car_distance", "
       not_exist_rear_car_distance","
       safe_distance_rear_car_distance","
       coherence_rear_car_distance");
    spec.clause("coherence_rear_car_distance", "
       not_exist_rear_car_distance", "
       safe_distance_rear_car_distance", "
       emergency_distance_rear_car_distance");
    spec.forbids("not_exist_rear_car_distance","
       safe_distance_rear_car_distance");
    spec.forbids("not_exist_rear_car_distance","
       emergency_distance_rear_car_distance");
    spec.forbids("safe_distance_rear_car_distance","
       not_exist_rear_car_distance");
    spec.forbids("safe_distance_rear_car_distance","
       emergency_distance_rear_car_distance");
    spec.forbids("emergency_distance_rear_car_distance","
       not_exist_rear_car_distance");
    spec.forbids("emergency_distance_rear_car_distance","
       safe_distance_rear_car_distance");
    spec.not("coherence_rear_car_distance");
  }

}
```

Listing A.21: MyNameMapper.java

```
package checksafety;

import java.util.HashMap;
import java.util.LinkedList;

import fr.kairos.lightccsl.core.stepper.INameToIntegerMapper;

public class MyNameMapper implements INameToIntegerMapper {
  private LinkedList<String> names = new LinkedList<>();
```

```
  private HashMap<String,Integer> nameToId = new HashMap<>();

  @Override
  public Iterable<String> getClockNames() {
    return names;
  }
  @Override
  public int getIdFromName(String name) {
    Integer v = nameToId.get(name);
    if (v == null) {
      nameToId.put(name, v = names.size());
      names.add(name);
    }
    return v;
  }
  @Override
  public String getNameFromId(int arg0) {
    return names.get(arg0);
  }
  @Override
  public int size() {
    return names.size();
  }


}
```

Listing A.22: Test.java

```
package checksafety;

import java.util.List;

import fr.kairos.lightccsl.core.stepper.ClockStatus;
import fr.kairos.lightccsl.core.stepper.ISolutionSet;
import fr.kairos.lightccsl.core.stepper.IStep;
import fr.kairos.timesquare.ccsl.sat.IBooleanSpecification;
import fr.kairos.timesquare.ccsl.sat.ISATSolverBuilder;
import fr.unice.lightccsl.sat.SAT4JSolverBuilder;

public class Test {
  public static void main(String[] args) {
    ISATSolverBuilder builder = new SAT4JSolverBuilder();
//    ISATSolverBuilder builder = new SATBDDBuilder();
    IBooleanSpecification boolSpec = builder.specification();
//    BooleanSatisfactionProblem bsp = (
   BooleanSatisfactionProblem)boolSpec;

    MyNameMapper mapper = new MyNameMapper();
    boolSpec.setNameMapper(mapper);
```

```
//Sat4jRules.build_goal2_cond1(boolSpec);
//Sat4jRulesConsistency.testParallelgoal2_cond1_goal3_cond1(
   boolSpec);
//Sat4jSystemConsistency.coherence_front_car_tracking(
   boolSpec);
//Sat4jSystemConsistency.coherence_straddling_car_tracking(
   boolSpec);
//Sat4jSystemConsistency.coherence_straddling_car_tracking(
   boolSpec);
//Sat4jSystemConsistency.coherence_goal2(boolSpec);
Sat4jSystemConsistency.system_solution(boolSpec);


//    ISolutionSet set = builder.solution();
//    try {
//    IStep step = set.pickOneSolution();
//    for (int i = 0; i< step.size(); i++) {
//      System.out.print(mapper.getNameFromId(i) + ":");
//      ClockStatus status = step.status(i);
//      switch(status) {
//      case Must: System.out.println("1, "); break;
//      case Cannot: System.out.println("0, "); break;
//      case May: System.out.println("?, "); break;
//      case Undetermined: System.out.println("x, "); break;
//      }
//    }
//    System.out.println(step);


ISolutionSet set = builder.solution();

try {
  List<? extends IStep> step = set.allSolutions();
  System.out.print(step+"\n");
  for (int j = 0; j<= step.size()-1; j++) {
    System.out.println("Solution "+(j +1)+":");
    System.out.println(step.get(j));
    IStep onestep = step.get(j);
    for (int i = 0; i< onestep.size(); i++) {
      System.out.print((i +1)+": " + mapper.getNameFromId(i)
         + "=");
      ClockStatus status = onestep.status(i);
      switch(status) {
      case Must: System.out.println("1, "); break;
      case Cannot: System.out.println("0, "); break;
      case May: System.out.println("?, "); break;
      case Undetermined: System.out.println("x, "); break;
      }
    }
    System.out.println("\n");
  }
```

```
      } catch (Exception e) {
        e.printStackTrace();
      }

    }
  }
```

Bibliography

[1] I. Standard, "26262 road vehicles-functional safety," *International Organization for Standardization, www. iso. org, date of access*, vol. 20171210, 2018.

[2] Eclipse. Xtext. Accessed: 2022-04-03. [Online]. Available: https://www.eclipse.org/Xtext/documentation/308_emf_integration.html

[3] A. Barišić, V. Amaral, and M. Goulão, "Usability driven dsl development with use-me," *Computer Languages, Systems & Structures*, vol. 51, pp. 118–157, 2018.

[4] C. in Colour. (2022) Understanding camera lens flare. Accessed: 2022-02-07. [Online]. Available: https://www.cambridgeincolour.com/tutorials/lens-flare.htm

[5] S. R. Center. (2022) Understanding camera lens flare. Accessed: 2022-04-06. [Online]. Available: https://www.trafficsafetystore.com/blog/winter-rain-and-fog-oh-my-is-autonomous-technology-ready-for-harsh-weather/

[6] L. Vanbever. (2019) Self-driving networks: Breaking new ground in network automation. Accessed: 2020-06-05. [Online]. Available: http://univ-cotedazur.fr/en/eur/ds4h/research/forum-numerica/forum-numerica/past-sessions/laurent-vanbever

[7] J. Cui, L. S. Liew, G. Sabaliauskaite, and F. Zhou, "A review on safety failures, security attacks, and available countermeasures for autonomous vehicles," *Ad Hoc Networks*, vol. 90, p. 101823, 2019, recent advances on security and privacy in Intelligent Transportation Systems. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1570870518309260

[8] C. A. Driving.eu. (2022) France takes lead on allowing automated driving on public roads. Accessed: 2022-03-31. [Online]. Available: https://www.connectedautomateddriving.eu/blog/france-takes-lead-on-allowing-automated-driving-on-public-roads/

[9] J. F. Drazkowski, R. S. Fisher, J. I. Sirven, B. M. Demaerschalk, L. Uber-Zak, J. G. Hentz, and D. Labiner, "Seizure-related motor vehicle crashes in arizona before and after reducing the driving restriction from 12 to 3 months," in *Mayo Clinic Proceedings*, vol. 78, no. 7. Elsevier, 2003, pp. 819–825.

[10] N. Kalra and S. M. Paddock, "Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability?" *Transportation Research Part A: Policy and Practice*, vol. 94, pp. 182–193, 2016.

[11] P. Koopman and M. Wagner, "Challenges in autonomous vehicle testing and validation," *SAE International Journal of Transportation Safety*, vol. 4, no. 1, pp. 15–24, 2016.

[12] S. Shalev-Shwartz, S. Shammah, and A. Shashua, "On a formal model of safe and scalable self-driving cars," *CoRR*, vol. abs/1708.06374, 2017. [Online]. Available: http://arxiv.org/abs/1708.06374

[13] X. Yan, S. Feng, H. Sun, and H. X. Liu, "Distributionally consistent simulation of naturalistic driving environment for autonomous vehicle testing," *arXiv preprint arXiv:2101.02828*, 2021.

[14] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM computing surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.

[15] E. Vacchi and W. Cazzola, "Neverlang: A framework for feature-oriented language development," *Computer Languages, Systems & Structures*, vol. 43, pp. 1–40, 2015.

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

[17] J. Gray, K. Fisher, C. Consel, G. Karsai, M. Mernik, and J.-P. Tolvanen, "Dsls: the good, the bad, and the ugly," in *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, 2008, pp. 791–794.

[18] J. D'Ambrosio and G. Soremekun, "Systems engineering challenges and mbse opportunities for automotive system design," in *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE, 2017, pp. 2075–2080.

[19] J. Duprez, "An mbse modeling approach to efficiently address complex systems and scalability," in *INCOSE International Symposium*, vol. 28, no. 1. Wiley Online Library, 2018, pp. 940–954.

[20] 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.

[21] 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.

[22] 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.

[23] H. A. Simon, *The sciences of the artificial*. MIT press, 1996.

[24] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS quarterly*, pp. 75–105, 2004.

[25] R. Wieringa, "Empirical research methods for technology validation: Scaling up to practice," *Journal of systems and software*, vol. 95, pp. 19–31, 2014.

[26] ——, "Design science as nested problem solving," in *Proceedings of the 4th international conference on design science research in information systems and technology*, 2009, pp. 1–12.

[27] 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.

[28] I. ISO, "26262: Road vehicles-functional safety," *International Standard ISO/FDIS*, vol. 26262, 2011.

[29] ——, "Pas 21448-road vehicles-safety of the intended functionality," *International Organization for Standardization*, 2019.

[30] G. V. Tepteris, "Autonomous vehicles: Basic concepts in motion control and visual perception," Ph.D. dissertation, University of the Aegean, 2020.

[31] T. M. OMBUDSMAN. (2022) What is an ncap rating? Accessed: 2022-02-10. [Online]. Available: https://www.themotorombudsman.org/knowledge-base/what-is-an-ncap-rating

[32] A. Hayek and J. Börcsök, "Safety chips in light of the standard iec 61508: survey and analysis," in *2014 International Symposium on Fundamentals of Electrical Engineering (ISFEE)*. IEEE, 2014, pp. 1–6.

[33] ENSEMBLE. Sotif iso24418.1. Accessed: 2022-03-27. [Online]. Available: https://platooningensemble.eu/storage/uploads/documents/2021/03/24/ENSEMBLE-D2.10_Iterative-process-documentation-and-Item-Definition_Final.pdf

[34] tutorialspoint. Requirement based testing. Accessed: 2022-04-02. [Online]. Available: https://www.tutorialspoint.com/software_testing_dictionary/requirements_based_testing.htm

[35] S. S. Shadrin and A. A. Ivanova, "Analytical review of standard sae j3016 taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles with latest updates," *Automobile. Doroga. Infrastruktura.*, no. 3 (21), p. 10, 2019.

[36] M. Krammer, P. Stirgwolt, and H. Martin, "From natural language to semi-formal notation requirements for automotive safety," SAE Technical Paper, Tech. Rep., 2015.

[37] Runestone Academy. Formal and natural languages. Accessed: 2022-04-03. [Online]. Available: https://runestone.academy/ns/books/published/thinkcspy/GeneralIntro/FormalandNaturalLanguages.html#formal-and-natural-languages

[38] W. E. McUmber and B. H. Cheng, "A general framework for formalizing uml with formal languages," in *Software Engineering, International Conference on.* IEEE Computer Society, 2001, pp. 0433–0433.

[39] C. Gomez, J. Deantoni, and F. Mallet, "Multi-view power modeling based on uml, marte and sysml," in *2012 38th Euromicro Conference on Software Engineering and Advanced Applications.* IEEE, 2012, pp. 17–20.

[40] J. Friedman, "Matlab/simulink for automotive systems design," in *Proceedings of the Design Automation & Test in Europe Conference*, vol. 1. IEEE, 2006, pp. 1–2.

[41] D. D. Ward and S. E. Crozier, "The uses and abuses of asil decomposition in iso 26262," in *7th IET International Conference on System Safety, incorporating the Cyber Security Conference 2012.* IET, 2012, pp. 1–6.

[42] S. Kent, "Model driven engineering," in *International conference on integrated formal methods.* Springer, 2002, pp. 286–298.

[43] OMG. (2022) Omg standards development organization. Accessed: 2022-04-12. [Online]. Available: https://www.omg.org/

[44] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework.* Pearson Education, 2008.

[45] A. G. Kleppe, J. B. Warmer, J. Warmer, and W. Bast, *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.

[46] I. Jacobson, "The use-case construct in object-oriented software engineering," in *Scenario-based design: envisioning work and technology in system development*, 1995, pp. 309–336.

[47] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, Tech. Rep., 1990.

[48] C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jezequel, "Automatic test generation: A use case driven approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 140–155, 2006.

[49] C. Atkinson and T. Kühne, "The role of metamodeling in mda," in *Proc. UML 2002 Workshop on Software Model Engineering*. Citeseer, 2002, pp. 67–70.

[50] S. Sendall and W. Kozaczynski, "Model transformation: The heart and soul of model-driven software development," *IEEE software*, vol. 20, no. 5, pp. 42–45, 2003.

[51] Borealis AI. Code generation (with xtend). Accessed: 2022-01-24. [Online]. Available: https://goto40.github.io/self\protect\discretionary{\char\hyphenchar\font}{}{}dsl/xtext_code_generation_xtend/

[52] GEMOC. (2022) The gemoc initiative. Accessed: 2022-04-10. [Online]. Available: https://gemoc.org/

[53] Eclipse. The easiest way to get your own modeling tool. Accessed: 2022-04-04. [Online]. Available: https://www.eclipse.org/sirius/

[54] N. Kahani and J. R. Cordy, "Comparison and evaluation of model transformation tools," *Queen's University, Kingston, Tech. Rep.*, 2015.

[55] B. Selic, "The pragmatics of model-driven development," *IEEE software*, vol. 20, no. 5, pp. 19–25, 2003.

[56] K. Czarnecki, "Overview of generative software development," in *International workshop on unconventional programming paradigms*. Springer, 2004, pp. 326–341.

[57] T. Stahl, M. Völter, and K. Czarnecki, *Model-driven software development: technology, engineering, management*. John Wiley & Sons, Inc., 2006.

[58] S. Cook, G. Jones, S. Kent, and A. C. Wills, *Domain-specific development with visual studio dsl tools*. Pearson Education, 2007.

[59] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi, "The generic modeling environment," in *Workshop on Intelligent Signal Processing, Budapest, Hungary*, vol. 17, no. 01. Citeseer, 2001.

[60] L. M. Rose, D. S. Kolovos, R. F. Paige, and F. A. Polack, "Model migration with epsilon flock," in *International conference on theory and practice of model transformations*. Springer, 2010, pp. 184–198.

[61] R. C. Gronback, *Eclipse modeling project: a domain-specific language (DSL) toolkit*. Pearson Education, 2009.

[62] A. Kleppe, *Software language engineering: creating domain-specific languages using metamodels*. Pearson Education, 2008.

[63] E. Visser, "Webdsl: A case study in domain-specific language engineering," in *International summer school on generative and transformational techniques in software engineering*. Springer, 2007, pp. 291–373.

[64] 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.

[65] 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.

[66] R. Passerone, D. Cancila, M. Albano, S. Mouelhi, S. Plosz, E. Jantunen, A. Ryabokon, E. Laarouchi, C. Hegedűs, and P. Varga, "A methodology for the design of safety-compliant and secure communication of autonomous vehicles," *IEEE Access*, vol. 7, pp. 125 022–125 037, 2019.

[67] 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.

[68] B. Jeannet and F. Gaucher, "Debugging embedded systems requirements with STIMULUS: an automotive case-study," in *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, Toulouse, France, Jan. 2016. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01292286

[69] Foretellix. (2020) Open measurable scenario description language (M-SDL). [Online]. Available: https://www.foretellix.com/open-language/

[70] A. Mammar, M. Frappier, and R. Laleau, "An event-b model of an automotive adaptive exterior light system," in *International Conference on Rigorous State-Based Methods*. Springer, 2020, pp. 351–366.

[71] J.-R. Abrial and A. Hoare, *The B-book: assigning programs to meanings*. Cambridge university press Cambridge, 1996, vol. 1.

[72] J.-R. Abrial, "Faultless systems: Yes we can!" *Computer*, vol. 42, no. 9, pp. 30–36, 2009.

[73] RATP. (2022) Ratp transportation. Accessed: 2022-04-12. [Online]. Available: https://www.ratp.fr/

[74] A. C. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh, "Inconsistency handling in multiperspective specifications," *IEEE Transactions on Software Engineering*, vol. 20, no. 8, pp. 569–578, 1994.

[75] B. Schatz, P. Braun, F. Huber, and A. Wisspeintner, "Consistency in model-based development," in *10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, 2003. Proceedings.* IEEE, 2003, pp. 287–296.

[76] S. J. Herzig, A. Qamar, and C. J. Paredis, "An approach to identifying inconsistencies in model-based systems engineering," *Procedia Computer Science*, vol. 28, pp. 354–362, 2014.

[77] A. Kreutzmann, D. Wolter, F. Dylla, and J. H. Lee, "Towards safe navigation by formalizing navigation rules," *TransNav : International Journal on Marine Navigation and Safety of Sea Transportation*, vol. Vol. 7, no. 2, pp. 161–168, 2013.

[78] J. Brunel, L. Rioux, S. Paul, A. Faucogney, and F. Vallée, "Formal safety and security assessment of an avionic architecture with alloy," *Electronic Proceedings in Theoretical Computer Science*, vol. 150, p. 8–19, May 2014. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.150.2

[79] Baidu. Appollo simulator user guide. Accessed: 2020-06-05. [Online]. Available: http://www.apollo.auto/index.html

[80] 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

[81] Five AI. Using simulation in the safety assurance of autonomous vehicles. Accessed: 2020-06-05. [Online]. Available: http://dsc2019.org/Docs/FiveAI-DSC2019-v15.pdf

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

[83] S. Außerlechner, S. Fruhmann, W. Wieser, B. Hofer, R. Spörk, C. Mühlbacher, and F. Wotawa, "The right choice matters! smt solving substantially improves model-based debugging of spreadsheets," in *2013 13th International Conference on Quality Software*. IEEE, 2013, pp. 139–148.

[84] J. Marques-Silva and S. Malik, "Propositional sat solving," in *Handbook of Model Checking*. Springer, 2018, pp. 247–275.

[85] J. Gray, S. Neema, J.-P. Tolvanen, A. S. Gokhale, S. Kelly, and J. Sprinkle, "Domain-specific modeling." *Handbook of dynamic system modeling*, vol. 7, pp. 7–1, 2007.

[86] 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.

[87] Eclipse. Code generation (with xtend). Accessed: 2022-04-04. [Online]. Available: https://www.eclipse.org/modeling/emf/

[88] L. Bettini, *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[89] P. Koopman, R. Hierons, S. Khastgir, J. Clark, M. Fisher, R. Alexander, K. Eder, P. Thomas, G. Barrett, P. Torr *et al.*, "Certification of highly automated vehicles for use on uk roads: Creating an industry-wide framework for safety," *White Rose Research Online*, 2019.

[90] J. E. Naranjo, C. González, J. Reviejo, R. García, and T. De Pedro, "Adaptive fuzzy control for inter-vehicle gap keeping," *IEEE Transactions on Intelligent Transportation Systems*, vol. 4, no. 3, pp. 132–142, 2003.

[91] Mozilla. Expressions and operators. Accessed: 2022-04-05. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators

[92] N. Chomsky, "Three models for the description of language," *IRE Transactions on information theory*, vol. 2, no. 3, pp. 113–124, 1956.

[93] K. Bengler, K. Dietmayer, B. Farber, M. Maurer, C. Stiller, and H. Winner, "Three decades of driver assistance systems: Review and future perspectives," *IEEE Intelligent transportation systems magazine*, vol. 6, no. 4, pp. 6–22, 2014.

[94] J. E. Stellet, M. R. Zofka, J. Schumacher, T. Schamm, F. Niewels, and J. M. Zöllner, "Testing of advanced driver assistance towards automated driving: A survey and taxonomy on existing approaches and open questions," in *2015 IEEE 18th International Conference on Intelligent Transportation Systems*. IEEE, 2015, pp. 1455–1462.

[95] A. Pütz, A. Zlocki, J. Bock, and L. Eckstein, "System validation of highly automated vehicles with a database of relevant traffic scenarios," *situations*, vol. 1, p. E5, 2017.

[96] J. Ploeg, E. de Gelder, M. Slavík, E. Querner, T. Webster, and N. de Boer, "Scenario-based safety assessment framework for automated vehicles," *arXiv preprint arXiv:2112.09366*, 2021.

[97] A. S. Mueller, J. B. Cicchino, and D. S. Zuby, "What humanlike errors do autonomous vehicles need to avoid to maximize safety?" *Journal of safety research*, vol. 75, pp. 310–318, 2020.

[98] A. Christensen, A. Cunningham, J. Engelman, C. Green, C. Kawashima, S. Kiger, D. Prokhorov, L. Tellis, B. Wendling, and F. Barickman, "Key considerations in the development of driving automation systems," in *24th enhanced safety vehicles conference. Gothenburg, Sweden*, 2015.

[99] V. Isler and R. Bajcsy, "The sensor selection problem for bounded uncertainty sensing models," in *IPSN 2005. Fourth International Symposium on Information Processing in Sensor Networks, 2005.* IEEE, 2005, pp. 151–158.

[100] A. Prudius, A. Karpunin, and A. Vlasov, "Analysis of machine learning methods to improve efficiency of big data processing in industry 4.0," *Journal of Physics: Conference Series*, vol. 1333, p. 032065, 10 2019.

[101] Y. Kondratenko, I. Atamanyuk, I. Sidenko, G. Kondratenko, and S. Sichevskyi, "Machine learning techniques for increasing efficiency of the robot's sensor and control information processing," *Sensors*, vol. 22, no. 3, p. 1062, 2022.

[102] L. Lazos and R. Poovendran, "Hirloc: High-resolution robust localization for wireless sensor networks," *IEEE Journal on selected areas in communications*, vol. 24, no. 2, pp. 233–246, 2006.

[103] J. D'Ambrosio, A. Adiththan, E. Ordoukhanian, P. Peranandam, S. Ramesh, A. M. Madni, and P. Sundaram, "An mbse approach for development of resilient automated automotive systems," *Systems*, vol. 7, no. 1, p. 1, 2019.

[104] Ansys. (2022) Why simulation is a driving force for autonomous vehicles. Accessed: 2022-02-07. [Online]. Available: https://www.ansys.com/blog/simulation-drives-autonomous-vehicles

[105] C. Hoyos, B. D. Lester, C. Crump, D. M. Cades, and D. Young, "Consumer perceptions, understanding, and expectations of advanced driver assistance systems (adas) and vehicle automation," in *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, vol. 62, no. 1. SAGE Publications Sage CA: Los Angeles, CA, 2018, pp. 1888–1892.

[106] Continental. (2022) Continental builds new plant for advanced driver assistance systems in the usa. Accessed: 2022-04-06. [Online]. Available: https://www.continental.com/en/press/press-releases/advanced-driver-assistance-systems/

[107] S. Schleicher, C. Gelau *et al.*, "The influence of cruise control and adaptive cruise control on driving behaviour–a driving simulator study," *Accident Analysis & Prevention*, vol. 43, no. 3, pp. 1134–1139, 2011.

[108] V. Sandner, "Development of a test target for aeb systems," in *23rd international technical conference on the enhanced safety of vehicles (ESV)*, no. 13-0406, 2013.

[109] S. Rasmana, D. Adiputra, W. Yahya, M. A. Rahman, A. Dwijotomo, M. M. Ariff, and N. A. Husain, "A systematic review on the autonomous emergency steering assessments and tests methodology in asean," *Journal of the Society of Automotive Engineers Malaysia*, vol. 5, no. 2, pp. 185–193, 2021.

[110] H. Hagras, V. Callaghan, M. Colley, and M. Carr-West, "A fuzzy-genetic based embedded-agent approach to learning and control in agricultural autonomous vehicles," in *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No. 99CH36288C)*, vol. 2. IEEE, 1999, pp. 1005–1010.

[111] V. Méndez, H. Catalán, J. R. Rosell, J. Arnó, R. Sanz, and A. Tarquis, "Simlidar–simulation of lidar performance in artificially simulated orchards," *Biosystems engineering*, vol. 111, no. 1, pp. 72–82, 2012.

[112] Deepdrive. (2022) Deepdrive self-driving ai. Accessed: 2022-02-08. [Online]. Available: https://deepdrive.io/

[113] Udacity. (2022) Udacity open sources its self-driving car simulator for anyone to use—techcrunch. Accessed: 2022-02-08. [Online]. Available: https://techcrunch.com/2017/02/08/udacity-open-sources-its-self-driving-car-simulator-for-anyone-to-use/?guccounter=2

[114] AirSim. (2022) Welcome to airsim. Accessed: 2022-02-08. [Online]. Available: https://microsoft.github.io/AirSim/

[115] Engadget. (2022) 'carcraft' is waymo's virtual world for autonomous vehicle testing. Accessed: 2022-02-08. [Online]. Available: https://www.engadget.com/2017-08-23-waymo-virtual-world-carcraft.html

[116] S. Bechtold and B. Höfle, "Helios: A multi-purpose lidar simulation framework for research, planning and training of laser scanning operations with airborne, ground-based mobile and stationary platforms." *ISPRS Annals of Photogrammetry, Remote Sensing & Spatial Information Sciences*, vol. 3, no. 3, 2016.

[117] SCANeR. AVSimulation User Guide for SCANeR. Accessed: 2020-06-08. [Online]. Available: https://www.avsimulation.com/solutions/

[118] M. Holen, K. Knausgård, and M. Goodwin, "An evaluation of autonomous car simulators and their applicability for supervised and reinforcement learning."

[119] F. Rosique, P. J. Navarro, C. Fernández, and A. Padilla, "A systematic review of perception system and simulators for autonomous vehicles research," *Sensors*, vol. 19, no. 3, p. 648, 2019.

[120] K. Jo, J. Kim, D. Kim, C. Jang, and M. Sunwoo, "Development of autonomous car—part i: Distributed system architecture and development process," *IEEE Transactions on Industrial Electronics*, vol. 61, no. 12, pp. 7131–7140, 2014.

[121] Y. S. Mahajan, Z. Fu, and S. Malik, "Zchaff2004: An efficient sat solver," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2004, pp. 360–375.

[122] B. Dutertre and L. De Moura, "The yices smt solver," *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, vol. 2, no. 2, pp. 1–2, 2006.

[123] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a sat-solver," in *International conference on formal methods in computer-aided design*. Springer, 2000, pp. 127–144.

[124] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[125] X. Gillard, C. Pecheur, S. BUSARD, and R. SADRE, "Adding sat-based model checking to the pynusmv framework," Ph.D. dissertation, Master's thesis, M. Sc. Thesis, Université Catholique de Louvain, 2016.

[126] B. König, M. Nederkorn, and D. Nolte, "Cores: a tool for computing core graphs via sat/smt solvers," *Journal of Logical and Algebraic Methods in Programming*, vol. 109, p. 100484, 2019.

[127] S. Malik and L. Zhang, "Boolean satisfiability from theoretical hardness to practical success," *Communications of the ACM*, vol. 52, no. 8, pp. 76–82, 2009.

[128] A. Gharbi, O. Fischer, and D. N. Mavris, "Towards a robust computational solution for the verification and validation of complex systems in mbse using wymore's tricotyledon theory of system design," in *AIAA SCITECH 2022 Forum*, 2022, p. 0094.

[129] F. Marić, "Formal verification of a modern sat solver by shallow embedding into isabelle/hol," *Theoretical Computer Science*, vol. 411, no. 50, pp. 4333–4356, 2010.

[130] C. A. Tovey, "A simplified np-complete satisfiability problem," *Discrete applied mathematics*, vol. 8, no. 1, pp. 85–89, 1984.

[131] K. Marussy, O. Semeráth, and D. Varró, "Automated generation of consistent graph models with multiplicity reasoning," *Submitted to the IEEE for possible publication*, 2020.

[132] D. Le Berre and A. Parrain, "The sat4j library, release 2.2," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 7, no. 2-3, pp. 59–64, 2010.

[133] F. Mallet, "ccsl–sts," urlhttps://github.com/frederic–mallet, 2021.

[134] V. L. Bernardin Jr, T. Mansfield, B. Swanson, H. Sadrsadat, and S. Bindra, "Scenario modeling of autonomous vehicles with trip-based models," *Transportation Research Record*, vol. 2673, no. 10, pp. 261–270, 2019.

[135] H.-C. Liu, L. Liu, and N. Liu, "Risk evaluation approaches in failure mode and effects analysis: A literature review," *Expert systems with applications*, vol. 40, no. 2, pp. 828–838, 2013.

[136] B. Vesely, "Fault tree analysis (fta): Concepts and applications," *NASA HQ*, 2002.

[137] ISO26262. System safety(functional safety, sotif, cyber security). Accessed: 2022-03-07. [Online]. Available: https://iso26262fs.wordpress.com/tag/hara/

[138] ISO 26262-1. (2011) Functional safety. Accessed: 2022-03-10. [Online]. Available: http://www.iso.org/

[139] F. Han, A. W. Bandarkar, and Y. Sozer, "Energy harvesting from moving vehicles on highways," in *2019 IEEE Energy Conversion Congress and Exposition (ECCE)*. IEEE, 2019, pp. 974–978.

[140] M. H. C. Torres, J.-P. Giacalone, and J. Abou Faysal, "A case study on formally validating motion rules for autonomous cars," in *International Conference on Software Engineering and Formal Methods*. Springer, 2020, pp. 233–248.

[141] IVEX. Intelligent vehicle technology: Enable safe autonomy. Accessed: 2022-03-10. [Online]. Available: https://www.ivex.ai/

[142] F. Mallet and R. De Simone, "Correctness issues on marte/ccsl constraints," *Science of Computer Programming*, vol. 106, pp. 78–92, 2015.

[143] D. Phung, "Implementation of graphical editor using sirius," 2018.