Safety analysis of inconsistencies using a formal verification tool for DSML

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

Safety Analysis of Inconsistencies Using a Formal Verification Tool for DSML

Joelle Abou Faysal[1,2] Nour Zalmai[1] Ankica Barisic[2]
Frederic Mallet[2]

(1) Renault Software Factory, Sophia Antipolis, France
nour.zalmai@renault.com, joelle.abou-faysal@etu.univ-cotedazur.fr
(2) Université Côte d'Azur, Cnrs, Inria, I3S, Sophia Antipolis, France
Ankica.Barisic,Frederic.Mallet@univ-cotedazur.fr

Abstract

Safety software engineers lack automatic interaction tools during Autonomous Vehicle development, which can help them check the correctness of safety rules, ensure the system's reliability in countless possible situations and its resistance to possible environmental conditions. In this paper, we discuss the benefits of the proposed approach to tackling these problems by enhancing safety software engineers to define formal safety requirements that follow one common format in the safety domain. Our framework is in charge of generating the verification engine to check inconsistencies and deploying a chosen solver. We discuss the necessity of using a SAT solver. The validity of the approach is illustrated in an application to a use case with a formal safety goal defined from a scenario. Our approach allows the safety software engineer to detect rule inconsistencies from his defined requirements and proceed to a modification.

keywords. Simulation, scenarios, model-based system engineering, autonomous vehicles, test and validation.

Introduction

Formal verification of safety methods is a complex task, but it is crucial in the field of autonomous vehicles (AVs). Their application has yielded good results and has increased the importance of system designers to use these technologies in their strategies. However, these approaches generally depend on formal languages specific to verification; thus, it is more difficult for designers to acquire. To enhance the development of safety systems, Model-Based Systems Engineering (MBSE) has been used to provide early validation and verification for Domain-Specific Modeling Languages (DSML). They help catch mistakes and effectively study solutions by formalizing specifications. They also clarify

concepts and paper proofs by complementing them with formal strategies. Multiple DSML approaches carry out software verification techniques and test the implementation by applying the Boolean Satisfiability Problem (SAT). The SAT is known for computational complexity, representing the first decision problem to be proved NP-complete ([12]). The SAT Solver has been used in many practical applications and is remarkably motivating MBSE. It serves as a trusted kernel checker for verifying the results of other untrusted verifiers such model checkers and Satisfiability Modulo Theories (SMT) solvers ([11]). An existing solution for safety checking has been introduced by ([14]), where the authors have developed a model-based for testing AVs' scenarios. The major drawback of their proposed solution is that it lacks formal description of the specifications for it can not be recognized whether their safety rules generation is sound or complete. Another work is provided by ([15]), where they expanded a model to study safety requirements. Unfortunately, their model lacks many environmental conditions such as weather, which makes it parametric. The need for automated vehicles (AVs) necessitates the establishment of a safety assessment methodology for road approval authorities following safety international standards to evaluate safety. Operational safety defined in the ISO 26262 standard ([7]) implies having a design for the safety component that deals with all unsafe situations. The safety of intended functionality (SOTIF) approach that extends ISO 26262, is defined in ISOPAR 21448.1 ([5]). 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 artefacts. This formalization process helps produce proof as required by ISO 26262 and accredits the goals in ASIL-D highly recommended by ISO 26262.

In this paper, we provide a DSML for AVs that allows domain experts to formally express all their safety requirements. The tool developed will autogenerate code that is fed to the SAT solver. This enables detecting inconsistencies and checking the soundness of the rules. If the user wishes to modify the environment, the code will adapt automatically to his alterations.

The main objectives of this approach are:

1. facilitating the formal description of user's safety rules,
2. auto-generating Java code necessary to detect inconsistencies that allow to modify his rules.

This paper is organized as follows. Section 1 presents relevant background information and the need for a SAT solver by giving a rule inconsistency example. Section 2 presents formal Extensible Platform for Safety Analysis of Autonomous Vehicle (EPSAAV) tool proposed using DSML. We begin by detailing the EPSAAV language development and presenting the user process where he is informed of the procedure to deploy this language. We mention the implementation steps in the tool to automatically generate Java code for the SAT solver. We show how this compatible code is then passed 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 of the rules. These inconsistencies can result from a requirement or specifications defined by the user. To assess that, we present in section 3 tests in an application of the EPSAAV tool to a Renault use case to verify SAT solutions. Finally, section 4 concludes and discusses future directions.

1 Background

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 ([16]). Recent breakthroughs in their development have also resulted 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 ([3]), verification of models, and use case generation ([4]). SMT solvers that solve more advanced theories are necessarily incomplete or even slower than SAT solvers ([8]). 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 paper, we pick out the suitable SAT solver and integrate it into the framework. The SAT solver gives us a certainty of the validity of our results. For future perspectives, SMT could be integrated to more complex cases. The formal rules defined are based on simple logical operations which can be defined using the SAT Solver.
The SAT solver addresses the satisfiability problem and is known for computational complexity, representing the first decision problem to be proven NP-complete ([12, 9]). It has been used in many practical applications and is remarkably motivating MBSE ([6]). Their proposed tool is highly accurate at finding inconsistencies by interpreting domain specifications as a logic problem ([13]).

We have a policy system in a Rule-Based Planner (RBP) having the following properties:

$$\begin{aligned} RBP = (&Goal1Condition1 \implies action1) \wedge \\ &(Goal1Condition2 \implies action1) \wedge \\ &(Goal1Condition3 \implies action1) \wedge \\ &(Goal2Condition1 \implies action2) \end{aligned} \tag{1}$$

with

$$\begin{aligned} &Goal1Condition1 = \text{follow the PV;} \\ &Goal1Condition2 = \text{respect speed threshold;} \\ &Goal1Condition3 = \text{respect safety distance of 2s;} \\ &\quad \text{and } action1 = \text{light acceleration.} \\ &Goal2Condition1 = \text{respect a Time To Collision (TTC);} \\ &\quad \text{and } action2 = \text{emergency braking.} \end{aligned} \tag{2}$$

Conflict occurs with RBP because it satisfies action1 and action2 (equation (1)). Satisfying them means that the Autonomous Vehicle (AV) is allowed to accelerate and decelerate at the same time. Imagine the case 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 have the truth table for the expression. 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. Our method requires deriving an explicit conflict by giving the possibility to execute rules in sequence (priority) or parallel. We then 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.

2 From safety rules to satisfiability rules using EPSAAV tool

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 1.

The requirement is translated 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 the EPSAAV tool to generate Java method bodies for operations specified in requirements. In our previous study, we detail the EPSAAV development part and describe the technologies used ([1]). EPSAAV tool is based on a meta-model that facilitates the environment's description and auto-generates monitors such as code and documents. In another work, we generate C code compatible with Renault's simulator, to study rule violation cases and to examine their generation ([2]). In this paper, we discuss the inconsistency study process, where we rely on multiple formulas adaptable to the SAT solver, presented and implemented from ([10]). We chose to integrate the code proposed for the ease of properties name mapping. If there is an inconsistency present, the EPSAAV tool can detect its origin and then asks the safety expert to modify the rules.

In other words, three undertakings are developed and auto-generated by the process:

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

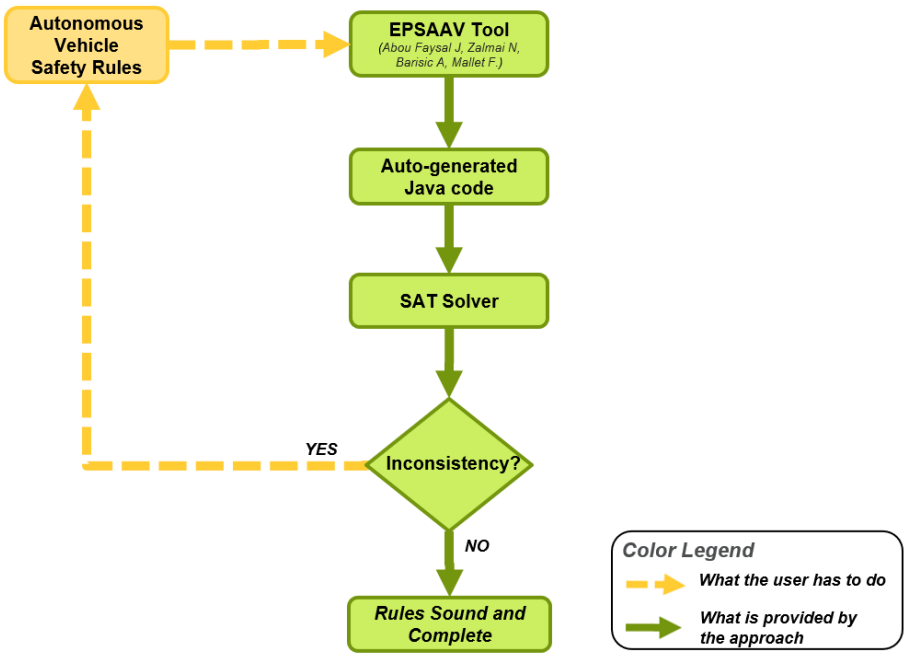

Figure 1: Inconsistency study process using *EPSAAV* Tool and the SAT Solver.

The third undertaking will be elaborated in our case study in section 3.

2.1 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. The RBP formula defined previously in equation (1) is an example of Boolean logic formulas with 6 defined variables. 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 auto-generating Java 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.

Figure 2 is a way of formalizing the specifications presented in equation (1) provided by EPSAAV tool. The safety expert is responsible for specifying safety rules and uses the EPSAAV tool to fill all his requirements. This example shows two rules in which he gives a priority (goalType) to the *emergency_braking* if there is a pedestrian, and then if not, we can perform a *light_acceleration* in

```
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 2: Definition of a goal with two sequence conditions containing specific logical operators.

case of a safe distance with the front car. This case refers to the *EgoProperties* library, defined by the safety expert and provided by EPSAAV tool, which contains four properties: *front_car_distance*, *front_car_tracking*, *pedestrian_distance* and *pedestrian_tracking* as shown in figure 3. Each property is assigned to states, and each state can be assigned to thresholds or variables. For example, the *safe_distance* is equal or greater to two seconds (*safety_distance* threshold). The goal translated into Boolean formulas is shown in figure 4. *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 that we internally define and auto-generate. This is how all the rules are generated using the EPSAAV tool and transformed into functions interfaced with the SAT4J.

```
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 3: Definition of an Ego property library with three states containing variables.

```
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 4: Java code auto-generation from rules defined by the safety expert in figure 2 compatible to the SAT4J library.

2.2 Applying inconsistencies tests

The second phase consists of testing the rules with specific formulas by automatically generating specific controls for each rule. When the safety engineer decides to write his code, he can make a mistake in the order of the rules or even repeat his definition twice. To help him determine whether his RBP is consistent or not, several tests should be run.

First test on each rule or condition defined by the engineer as follows:

$$test(R) \begin{cases} true, & \text{rule valid} \\ false, & \text{rule not valid} \end{cases} \tag{3}$$

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. It is also important

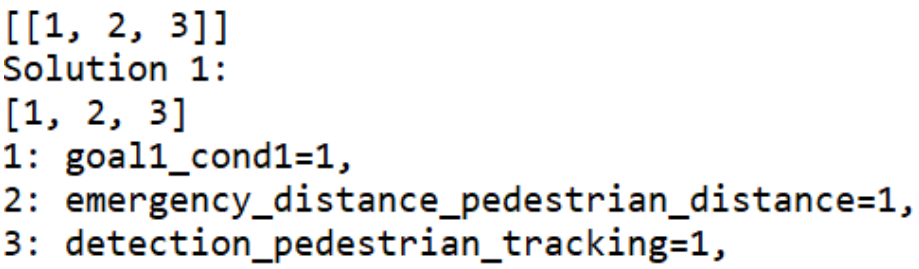

Figure 5: A solution for the auto-generated *build_goal1_cond1_True()*.

to be able to pass by the solutions and see if there is a case in the truth table where two actions can be triggered at the same time in case of multiple conditions in one rule. In order to see solutions for this rendered objective, we need to force the rules to become true. To study the validity of the internal rules, we automatically generate for each condition a true function. For example, for goal1_cond1, we have one solution for the auto-generated *build_goal1_cond1_True()* as shown in figure 5. The numbers in the tables of the figure 5 represent the number of variables which in our case is 3 variables. 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 or one. In this case, they are all positive since it is an AND operator.

Another test is applied on goals or even on two conditions within the same goal to determine the type of execution. These tests are also auto-generated by the EPSAAV tool. To test this inconsistency, we follow tests in equation (4). If R1 is executed before R2, we apply $\neg$R1$\vee$R2. 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.

$$test(R1, R2) \begin{cases} \neg\text{R2}\wedge\text{R1}, & \text{R2 prior to R1} \\ \neg\text{R1}\wedge\text{R2}, & \text{R1 prior to R2} \\ \text{R1} \wedge \text{R2}, & \text{parallel execution} \\ (\text{R1} => \text{R2}) \wedge (\text{R2} => \text{R1}), & \text{identical rules} \end{cases} \tag{4}$$

If conditions 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 contradictory states that are present in the requirement can be executed at the same time. For instance, we take the same example given in equation (1), formalized in figures 2 and 3, and auto-generated in functions presented in figure 4. If the GoalType is set to Constraint which means parallel, by testing parallel execution in equation (4) that is condition1 $\wedge$ condition2, we will have both actions *emergency_braking* and *light_acceleration* set to true. The system will alert the user that both actions could not be triggered together and there is an inconsistency. The safety engineer will then modify his requirements and check all possible solutions when the GoalType is reset to priority.

Another way to test consistency is to try to parse the solutions to the rule set. 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:

$$\begin{aligned} \textbf{R1 priority over R2 priority over R3} = \\ R1 \vee ((\neg R1) \wedge R2) \\ \vee(\neg(R1 \wedge R2) \wedge R3) \end{aligned} \tag{5}$$

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

$$\begin{aligned} \textbf{R1 parallel with R2 parallel with R3} = \\ R1 \vee R2 \vee R3 \end{aligned} \tag{6}$$

The coherence of the system is the coherence of all the rules with their behaviors. The system solutions are equal to coherence_all_goals_behaviors and coherence_all_properties as seen in the following equation (7):

$$\begin{aligned} \textbf{System\_solution} = \\ coherence\_all\_goals\_behaviors \\ \wedge coherence\_all\_properties \end{aligned} \tag{7}$$

The *coherence_all_goals_behaviors()* function uses an OR logic operator for all goals coherences. The *coherence_all_goals_properties()* function uses a XOR logic operator for all properties coherences because one property can only have one state true. For instance in figure 3, *pedestrian_distance* can either have *emergency_distance* or *not_exist* but not both at the same time.

To sum up, EPSAAV auto-generates a verification engine that contains three Java files:

- *Sat4jRules.java* that contains goals expressed in boolean specification functions,
- *Sat4jRulesConsistency.java* that forces the auto-generated functions in the previous file to be true and false, and shows the auto-generated code to test priority or parallel executions,

- and *Sat4jSystemConsistency.java* that tests all solutions for the RBP with the coherence of goals and properties.

All these tests are automatically generated by the EPSAAV tool, so the safety expert will not have to deal with development but instead will have automatic results and alerts on rule inconsistencies to help him define better requirements.

3 Application of EPSAAV approach to a use case

This section illustrates the EPSAAV framework with a case study on a Renault project that assessed known unsafe scenarios. The scenario, measures, and risks are formalized in goals in which we take one goal to check inconsistencies.

We take a scenario represented in figure 6 where Ego Vehicle (EV) is on the highway in a traffic jam. On the left and right sides of EV, there are other road users such as Straddling Vehicle (SV) or guardrails. 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. The risk is to have a

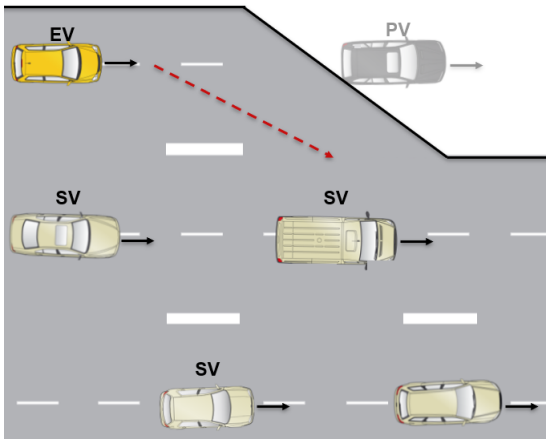

Figure 6: Scenario of an EV making a cut-in and does not detect the guardrail due to an invisible old line.

lateral collision with the SV. One of the measures to avoid a collision with the SV is that the system shall detect the incoherence between sensors that cause loss of obstacle detection. This is why we introduce a *bug_notification* alert that helps to study the bug from sensor perception.
The formalized rule that he has to insert in EPSAAV tool to be sure that there is no interference in the front car detection is the following: 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 no perception error. Figure 7 defines the opposite of these defined requirements using the NOT operator in order to trigger the *bug_notification* alert.

```
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 7: Goal1 in the RBP to detect interferences of PV and SV.

We obtain from EPSAAV the auto-generated Java code that helps us visualize solutions of *goal1*. We then apply the tests described in section 2 and proceed to the first test in equation (3). We obtain 16 solutions that constitute the possible cases where we can have a *bug_notification*. Figure 8 shows a solution where *front_car_tracking* (variable 7) and *straddling_car_tracking* (variable 11) are not compatible with *front_car_distance* (variable 4) and *straddling_car_distance* (variable 10) respectively, which cause triggering the specified alert.

After analyzing the 16 solutions, we notice 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* with the AND operator that includes the conflict in each obstacle type.

Figure 9 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 following the equation (7). We test *goal1_cond1* with the coherence of *front_car_tracking* property that only forces one of its states to be

```
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 8: Solution for **goal1_cond1** presenting an inconsistency in distance and tracking variables.

```
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 9: Solution for **goal1_cond1** presenting an inconsistency in the states of the same property.

true. This helps us deleting unreasonable cases (figure 10). This will improve tests and eliminate inconsistent solutions considering properties coherence.

4 Conclusion

In this paper, we discuss the benefits of the EPSAAV tool in helping safety engineers to define safety requirements. Our framework is in charge of auto-generating all the code to check inconsistencies and deploy a solver. The SAT solver is sufficient to prove our point and has enough expressiveness for the kind of language used in safety requirements. We present the three auto-generated tests to deploy the SAT solver. First, a Java generation is performed according to the safety rules defined by the expert. This generation follows a

```
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,
```

Figure 10: Solution for **goal1_cond1** that considers the coherence of **front_car_tracking** property.

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. In a second step, inconsistency tests on the rules and the system are carried out. We detail the generated Java verification engine to study the inconsistencies between the rules and the system. This helps them save development time and cost. 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 dig deeper into the application of safety requirements for a Renault use case to assess the usability of our approach in an industrial case study. For future work, we will use EPSAAV for more complex scenarios, we will deploy an SMT solver that can better deal with this complexity, and we will analyze the possible outputs.

References

[1] ABOU FAYSAL, J., ZALMAI, N., BARISIC, A., AND MALLET, F. Epsaav: An extensible platform for safety analysis of autonomous vehicles. In *International Conference on Model and Data Engineering* (2021), Springer, pp. 101–111.

[2] ABOU FAYSAL, J., ZALMAI, N., BARISIC, A., AND MALLET, F. Adaptation of an auto-generated code using a model-based approach to verify functional safety in real scenarios. In *ERTS 2022-Embedded Real Time Systems* (2022).

[3] AUSSERLECHNER, S., FRUHMANN, S., WIESER, W., HOFER, B., SPORK, R., MUHLBACHER, C., AND WOTAWA, F. The right choice matters! smt

solving substantially improves model-based debugging of spreadsheets. In *2013 13th International Conference on Quality Software* (2013), IEEE, pp. 139–148.

[4] DE MOURA, L., AND BJORNER, N. Satisfiability modulo theories: introduction and applications. *Communications of the ACM 54*, 9 (2011), 69–77.

[5] ENSEMBLE. Sotif iso24418.1. accessed: 2022-03-27.

[6] GHARBI, A., FISCHER, O., AND MAVRIS, D. N. 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.

[7] GOSAVI, M. A., RHOADES, B. B., AND CONRAD, J. M. Application of functional safety in autonomous vehicles using ISO 26262 standard: A survey. In *SoutheastCon 2018* (2018), IEEE, pp. 1–6.

[8] KONIG, B., NEDERKORN, M., AND NOLTE, D. Cores: a tool for computing core graphs via sat/smt solvers. *Journal of Logical and Algebraic Methods in Programming 109* (2019), 100484.

[9] MALIK, S., AND ZHANG, L. Boolean satisfiability from theoretical hardness to practical success. *Communications of the ACM 52*, 8 (2009), 76–82.

[10] MALLET, F. ccsl-sts. https://github.com/frederic-mallet/ccsl-sts, 2021.

[11] MARIĆ, F. Formal verification of a modern sat solver by shallow embedding into isabelle/hol. *Theoretical Computer Science 411*, 50 (2010), 4333–4356.

[12] MARQUES-SILVA, J., AND MALIK, S. Propositional sat solving. In *Handbook of Model Checking*. Springer, 2018, pp. 247–275.

[13] MARUSSY, K., SEMERÁTH, O., AND VARRÓ, D. Automated generation of consistent graph models with multiplicity reasoning. *Submitted to the IEEE for possible publication* (2020).

[14] SCHÜTT, B., BRAUN, T., OTTEN, S., AND SAX, E. 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.

[15] SHALEV-SHWARTZ, S., SHAMMAH, S., AND SHASHUA, A. On a formal model of safe and scalable self-driving cars. *arXiv preprint arXiv:1708.06374* (2017).

[16] SHEERAN, M., SINGH, S., AND STAALMARCK, G. Checking safety properties using induction and a sat-solver. In *International conference on formal methods in computer-aided design* (2000), Springer, pp. 127–144.