Journal of
Sensor and Actuator Networks

MDPI

Article

A Double-Level Model Checking Approach for an Agent-Based Autonomous Vehicle and Road Junction Regulations

Gleifer Vaz Alves [1,*], Louise Dennis [2] and Michael Fisher [2]

[1] Graduate Program in Computer Science (PPGCC), Federal University of Technology—Parana (UTFPR), Ponta Grossa 84017-220, PR, Brazil
[2] Department of Computer Science, University of Manchester, Manchester M13 9PL, UK; louise.dennis@manchester.ac.uk (L.D.); michael.fisher@manchester.ac.uk (M.F.)
* Correspondence: gleifer@utfpr.edu.br or gleifervaz@gmail.com

Abstract: Usually, the design of an Autonomous Vehicle (AV) does not take into account traffic rules and so the adoption of these rules can bring some challenges, e.g., how to come up with a Digital Highway Code which captures the proper behaviour of an AV against the traffic rules and at the same time minimises changes to the existing Highway Code? Here, we formally model and implement three Road Junction rules (from the UK Highway Code). We use timed automata to model the system and the MCAPL (*Model Checking Agent Programming Language*) framework to implement an agent and its environment. We also assess the behaviour of our agent according to the Road Junction rules using a double-level Model Checking technique, i.e., UPPAAL at the design level and AJPF (*Agent Java PathFinder*) at the development level. We have formally verified 30 properties (18 with UPPAAL and 12 with AJPF), where these properties describe the agent's behaviour against the three Road Junction rules using a simulated traffic scenario, including artefacts like traffic signs and road users. In addition, our approach aims to extract the best from the double-level verification, i.e., using time constraints in UPPAAL timed automata to determine thresholds for the AVs actions and tracing the agent's behaviour by using MCAPL, in a way that one can tell when and how a given Road Junction rule was selected by the agent. This work provides a proof-of-concept for the formal verification of AV behaviour with respect to traffic rules.

Keywords: Rules of the Road; Autonomous Vehicles; agents; model checking

Citation: Alves, G.V.; Dennis, L.; Fisher, M. A Double-Level Model Checking Approach for an Agent-Based Autonomous Vehicle and Road Junction Regulations. *J. Sens. Actuator Netw.* **2021**, *10*, 41. https://doi.org/10.3390/jsan10030041

Academic Editors: Claudio Savaglio, Daniela Briola, Rafael C. Cardoso, Angelo Ferrando, Claudio Menghi and Tobias Ahlbrecht

Received: 15 March 2021
Accepted: 21 June 2021
Published: 25 June 2021

1. Introduction

The deployment of Autonomous Vehicles (AVs) in urban road networks is possible in the near future. However, many challenges arise on the way towards this goal, for example: how can policy-makers ensure an AV is safe to operate within their jurisdiction [1]? This, and other complex issues, mean that the design and development of an AV must go through several stages involving a multistakeholder approach, which includes regulators, AV developers, safety experts, members of the public among others.

While the design of an AV should include sensors, cameras, software development, security protections, etc., it should also take into consideration the assessment of the traffic rules within which the AV will operate. If not, questions concerning safe operation cannot be properly answered. However, as highlighted by both Prakken [2] and Alves et al. [3,4] these traffic rules are rarely considered in the design and assessment of AVs.

1.1. Autonomous Vehicles and the Rules of the Road

Recent documents, such as [1,5,6] have started to highlight and discuss the need for a Digital Highway Code, where an AV would need to comply with the local traffic rules (or the "Rules of the Road"). It is well known that such a task brings challenges, mainly since Highway Codes were not designed to operate alongside autonomous sys-

tems, but also since the description of the rules is predominantly human-readable, and not machine-readable.

So, how can we tackle such challenges? On the one hand, an AV should comply with traffic laws in a way that requires very few changes to create a Digital Highway Code [1]. On the other hand, it is understandable that new "Rules of the Road" may need to be designed in order to properly handle autonomous systems in urban traffic environments [6]. There is a clear trade-off between these two issues. Two examples illustrate the need to have "Rules of the Road" designed into AVs. As highlighted in ref. [1], in the state of Arizona (US) an AV operator may be issued a citation if the AV does not comply with traffic laws. A further example can be seen in the PAS-1882 standard from the BSI [7], which specifies "data collection and management for automated vehicle trials". In this standard several mechanisms are described for collecting the necessary data to conduct an AV trial. To the best of our knowledge, this standard currently does not contain data concerning questions such as when and how the "Rules of the Road" have been used by the AV and these could be quite important. These two examples remind us that, by ensuring the autonomous software abides by the "Rules of the Road" when AV is designed, it is definitely a useful asset for the stakeholders concerned with the proper behaviour of an AV on the roads.

In ref. [8], Waymo released a Safety Report on their vehicles. This document presents a reference from the *National Highway Traffic Safety Administration* (NHTSA), which shows the four scenarios that accounted for the vast majority of crashes in the US:

- 29% of the vehicles were involved in rear-end crashes;
- 24% of the vehicles were turning or crossing at intersections just before the crashes;
- 19% of the vehicles ran off the edge of the road;
- 12% involved vehicles changing lanes.

Consequently, Road Junction rules (which deal with crossing and entering intersections) provide a good case study for understanding the interaction of AVs and the Rules of the Road. This will enable us to develop an approach that can inform stakeholders around the development of Digital Highway Codes.

In our work, we select three Road Junction rules, from the UK Highway code [9], because road junction behaviour is a contributory factor in many crashes as discussed above [8]. We aim to embed these traffic rules into an agent, where this agent describes the basic behaviour of an AV in Road Junction scenarios. With this, we intend to determine: (i) Can these three selected road junction rules be used directly (i.e., as seen in the UK Highway code) by an AV? (ii) How to assess the AVs behaviour against the three road junction rules considering simple Road Junction scenarios? and (iii) Are there any guidelines that can be given to enable the AV to work correctly with such Road Junction rules?

1.2. Related Work

Considering the related work, there are Rizaldi et al. [10] and Bhuiyan et al. [11] that present a formalisation for road traffic rules. Nonetheless, neither approach uses an agent abstraction to represent the AV behaviour and decision-making. Besides, Kamali et al. [12] and Al-Nuaimi et al. [13] both present the use of agents to model the AV and the formal verification of agents. However, their AV application scenario is not related to the "Rules of the Road". So, our work aims to formalise the Road Junction rules, use an agent to represent an AV and apply formal verification techniques to properly assess an agent's behaviour in road traffic scenarios .

In ref. [14], Bakar and Selamat present a systematic literature review of agent systems verification, where they describe the most used techniques to formally verifying agents. Their figures show that 49% of techniques are applied at the design stage, 27% during development, and 25% at runtime. Model Checking or model-based verification techniques are used in 44% of the work, while most of the properties verified are temporal ones (19%), followed by epistemic properties (9%). These figures serve to endorse our choice of a

double-level Model Checking, i.e., applying formal verification of temporal properties at both design and development levels.

1.2.1. Proposal

Figure 1 shows our proposed SAE-RoR (*Simulated Automotive Environment for the Rules Of the Road*) architecture. In previous work [4], we presented the first version of SAE-RoR architecture, where we focused on the formalisation of the Road Junction rules using Linear Temporal Logic (LTL) and also the first steps towards the implementation of a single rule using the agent programming language, GWENDOLEN [15].Now, we extend the SAE-RoR architecture by adding an extra layer of modelling with timed automata and Model Checking using UPPAAL [16]. We have also added two further Road Junction rules and the formal verification of properties using AJPF, which is responsible for Program Model Checking of the GWENDOLEN implementation [17].

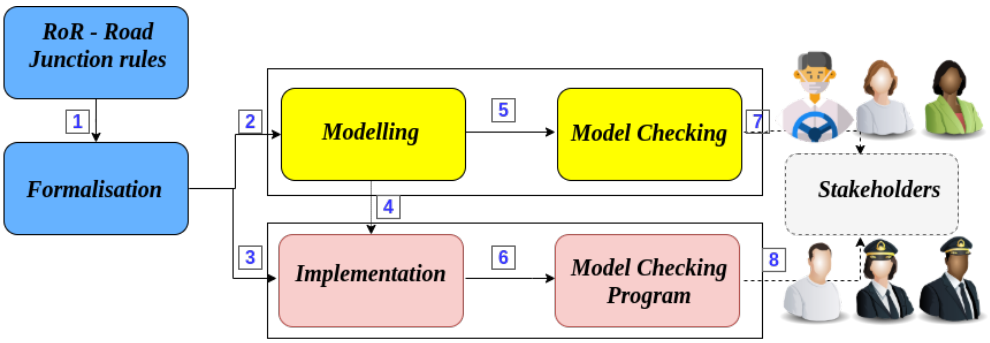

Figure 1. Proposal: SAE-RoR architecture.

Each element of the SAE-RoR architecture is described in detail in Sections 4–6. Here, we explain the general workflow using the architecture in Figure 1. Step 1, i.e., arrow 1 between the two blue components in Figure 1, represents the formalisation using LTL of the Road Junction rules from the UK Highway Code (this was initially described in [4]). This LTL formalisation helps us to abstract the informal concepts and elements in the UK Highway Code to an unambiguous formal language. Then steps 2 and 3 represent the use of this language as a basis to respectively build the UPPAAL model and the GWENDOLEN implementation. Notice that step 4 shows the mapping from UPPAAL model elements to the agent's implementation components. Next, step 5 represents the formal verification of properties of the model using the UPPAAL model checker, while step 6 shows those properties concerned with the agent's implementation that are verified using the AJPF model checker. Steps 7 and 8 describe stages of the SAE-RoR architecture that are not implemented yet (forming future work). Nonetheless, this Stakeholders stage is an important feature of our proposal. Steps 7 and 8 represent the outcomes from property verification that could guide the actions of a given Stakeholder. As an example, a Policymaker could use a counter-example describing a safety violation from a given model and decide whether a traffic rule concerning pedestrians needs to be changed. In ref. [6], some possible Stakeholders (related to AVs) are mentioned: Driver, Road User, Safety Expert, AV developer, Researcher, Policymaker, Traffic Officer, Emergency services and police, Local government, Highway authorities, Public sector, Insurance, Politicians, Legal, among others. These stakeholders form suitable end-users for our proposed workflow.

1.2.2. Contributions

The main contributions of our work are the following:

1. A complete architecture for the Formal Verification of Agents in the Rules of the Road, which starts at the formalisation of Road Junction rules, passes these to modelling and implementation tools, generates formal verification results, which may be of interest to the given stakeholders.

2. The double-level Model Checking approach, which makes it possible to formally verify properties both at design and development levels. As proof-of-concept, we have verified 30 properties, 18 at design level and 12 at development level.
3. A set of verified properties, where properties range from time constraints to the analysis of the AV-agent's behaviour considering all possible actions the agent can take in Road Junction scenarios.
4. The creation of an agent's environment that includes random generation of events (following the methodology outlined in ref. [18]), where different scenarios concerning selected Road Junction rules are simulated.
5. The use of a mapping from a given UPPAAL timed automaton to the basic elements of a BDI (Belief-Desire-Intention) agent.
6. Implementation of a BDI agent (in GWENDOLEN), which enables tracing of an agent's behaviour. Taken with the model-checking process (via AJPF) it is possible to assess what were the choices selected (autonomously) by the agent that led to any given outcome [19]. For example, given a certain scenario with specific perceptions from the environment, did the agent (correctly) choose to follow a given road junction rule?

1.2.3. Remarks and Limitations

It is necessary to determine some remarks and assumptions about our proposal, making it clear what is included (or not) in our model and implementation. Further, some limitations of our work are mentioned below.

- Single Agent: we only model a single agent in our implementation. And this agent has no (internal) concurrency to try to enter the Road Junction.
- No driving behaviour: this single agent has no driving behaviour component, the agent is only concerned with obeying the programmed Road Junction rules. So, we do not verify, for example, the speed or trajectory of the vehicle.
- No collision-freedom: we do not model collision avoidance behaviour here, though we have considered it in previous work [20,21].
- Intersection Management: we are not trying to deal with the well-known problem of Intersection Management using multiple agents [22]. We consider only the behaviour of a single agent following the desired traffic rules.
- Road Junction environment: The environment is represented in both modelling and simulation as a simple (9×9) grid with a few road features such as `stop signs`, and other road users. This is because this captures the abstract issues and, on a practical level, model-checking does not scale well once the grid size increases dramatically. Our environment model uses as basis the formalised abstract model which captures the temporal elements from the road junction rules. So, we do not represent spatial elements, as seen, for example in [23], where the author uses the Multi-lane Spatial Logic to represent virtual lanes in an urban traffic environment.
- Time constraints: we use time constraints in our model to represent thresholds within or after which actions and events should occur. These constraints are only used in our model, they are not taken from any Highway Code. This illustrates the way modellers need to extract implicit assumptions from rules written for human consumption—in this case that actions such as waiting will take a reasonable amount of time, or will take place within a reasonable space of time.
- Subset of Road Junction Rules: the UK Highway code has a set of 14 rules for Road Junctions. Here, we model and implement only the first three rules from the Road Junction set, specifically rules 170, 171, and 172.
- Modularity of the model checking stages: our two Model Checking stages (using UPPAAL and MCAPL) are loosely coupled, i.e., they take place independently of each other. On the one hand, this independence loses the potential benefit an integrated model checking semantics, where one could verify the whole system (model plus implementation). On the other hand, this modular architecture allows a separation of

concerns meaning a user can consider separately the verification of either the timed automata model verification or the agent's implementation.
- Nature of verified properties: all the verified properties are related to the basic behaviour of our agent against the three selected road junction rules in our simple road junction environment.

In our previous publications [3,4], we have presented the formalisation of the Road Junction rules in LTL and a partial implementation of Rule 170 in the GWENDOLEN language. Thus, this paper extends these initial results by the following:

- Including two new rules (171 and 172) from the UK Highway Code.
- Modelling the AV-agent and Road Junction environment using timed automata.
- Formal Verification of 18 properties of these timed automata using UPPAAL.
- Implementation of all three rules in GWENDOLEN.
- Formal Verification of 12 properties of this agent implementation using AJPF.

The remainder of this paper presents, in Sections 2 and 3, some useful terminology and background information. Next, Sections 4–6 describe the main stages of our work on modelling, implementation and formal verification results. In Section 7, related work is discussed while Section 8 provides final remarks.

2. Key Terminology & the Rules of the Road

In this section, we show the Road Junction rules that are used in our work. Before proceeding, we clarify some terms applied here to guide the description of our modelling, implementation, and verification.

2.1. Terminology and Abbreviation

- SAE-RoR: is the name of our proposed architecture (seen in Figure 1).
- AV: According to Herrmann et al. [24], the term automated vehicles refers to autonomy levels 1–4, while the terms autonomous, self-driving or driverless vehicles refer to autonomy level 5. Here, for the sake of simplicity, we only use the term Autonomous Vehicles (AV). And in our model, we are not concerned whether our agent represents a vehicle with level 4 or 5, or if the vehicle has a human driver responsible or only passengers inside it.
- AV-agent: is the name of our agent implemented in GWENDOLEN.
- `AV_agent`: is the name of the automaton modelled with UPPAAL, which represents the AV-agent.
- RU: according to the UK Highway Code, a Road User (RU) can be any of the following: pedestrian, cyclist, motorcyclist, powered wheelchair/mobility scooter, horse rider, etc.
- `ru`: control variable that represents a Road User and it is used in the UPPAAL timed automaton model.
- RJ: here we use the term Road Junction (RJ) which has the same meaning as an Intersection.
- RoR: we use the term "Rules of the Road" (RoR) which has the same meaning as traffic rules, Highway Code, traffic laws, or road traffic laws.
- Digital Highway Code: is the version of the Rules of the Road which is intended to work for AVs.
- Cross junction: a crossroad is the place where two roads meet and cross each other. It could be in the form of: a major road crossing a minor road; or two equal roads crossing each other [25].
- T junction: is a place where two roads meet in the shape of letter T [25].

2.2. The Road Junction Rules

The UK Highway Code has different sections, concerning Overtaking, Roundabouts, Road Junctions, among others [9]. We are focused on the Road Junction rules, which

has 14 rules, from 170 to 183, describing when and how a driver is supposed to enter a road junction, to turn right, to turn left, to enter a road junction with traffic lights, among other situations. Here, we describe the three simple Road Junction rules that are modelled, implemented, and verified, rules: 170, 171 and 172. Before presenting the formalisation of these three rules we describe the LTL (Linear Temporal Logic) [26] operators and constants that we use in our formalisation for the Road Junction rules. Further details about this formalisation can be found in Alves et al. [4].

- Propositional operators from LTL:
  $\wedge$, $\vee$, $\rightarrow$, $\neg$.

  where these four propositional logical operations represent conjunction, disjunction, implication, and negation.

  $\Box$, $\Diamond$, $\bigcirc$, $\cup$.

  where these four future-time operators represent: always, eventually, next, and until.

2.2.1. LTL Formalisation

- Rule 170—UK Highway Code:
  - You should watch out for road users (RU).
  - Watch out for pedestrians crossing a road junction (JC) into which you are turning. If they have started to cross they have priority, so give way.
  - Look all around before emerging (NB: For the sake of clarity, we choose to use the term enter as an action which represents not only a driver entering a road junction, but also emerging from a road junction to another road). Do not cross or join a road until there is a safe gap (SG) large enough for you to do so safely.
- Rule 170, represented in LTL, describes when the autonomous vehicle (AV) may enter the junction (JC):

$$\Box\,((\texttt{watch(AV, JC, RU)} \wedge (\neg\,\texttt{cross(RU, JC)} \wedge (\texttt{exists(SG, JC)})) \rightarrow ((\texttt{exists(SG, JC)} \wedge \neg\,\texttt{cross(RU, JC)}) \cup \texttt{enter(AV, JC)})))$$

  Informal Description: it is always the case that the AV is supposed to watch for any road users (RU) at the junction (JC) and there are no road users crossing the junction and there is a safe gap (SG). Then, no road users crossing the junction and the existence of a safe gap should remain true, until the AV may enter the junction.

- Rule 170 represented in LTL, when the autonomous vehicle (AV) should give way at the junction (JC):

$$\Box\,(\texttt{watch(AV,JC,RU)} \wedge (\texttt{cross(RU,JC)}) \rightarrow \texttt{give-way(AV,JC)})$$

  Informal Description: it is always necessary to watch out for road users (RU) and check if there is a road user crossing the junction. Then, the AV should give way to traffic.

- Rule 171—UK Highway Code:
  - You MUST stop behind the line at a junction with a 'Stop' sign (ST) and a solid white line across the road. Wait for a safe gap (SG) in the traffic before you move off.
- Rule 171 represented in LTL:

$$\texttt{exists(ST,JC)} \rightarrow \Box\,(\texttt{stop(AV,JC)} \cup (\texttt{exists(SG,JC)} \wedge (\texttt{exists(SG,JC)} \cup \texttt{enter(AV,JC)})))$$

`Informal Description`: when there is a stop sign (`ST`), then it is always the case the `AV` should stop at the junction until there is a safe gap (`SG`). And the safe gap must remain true until the `AV` enter at the junction.

- Rule 172—UK Highway Code:
  - The approach to a junction may have a 'Give Way' sign (`GW`) or a triangle marked on the road (`RO`). You MUST give way to traffic on the main road (`MR`) when emerging from a junction with broken white lines (`BWL`)across the road.
- Rule 172 represented in LTL:

$$\Box\ (\texttt{(exists(AV,RO)} \wedge \texttt{enter(AV,JC))}$$
$$\wedge\ (\texttt{(exists(BWL,JC)} \vee \texttt{exists(GW,JC))} \rightarrow \texttt{give-way(AV,MR)))}$$

`Informal Description`: It is always the case that when there is an `AV` driving on a Road (`RO`) and the `AV` enters the junction. And there is a Broken White Line (`BWL`) or a Give Way sign (`GW`), then the `AV` should give way to the traffic on the Main Road (`MR`).

2.2.2. Remarks

The LTL formalisation aims to describe most of the elements from the rules as given in the UK Highway Code, but some level of abstraction is needed to properly determine the formalisation. And when we build the automata models in UPPAAL and also the GWENDOLEN implementation of the AV-agent we have abstracted some additional elements from the formalised Road Junction rules, in a way that the three rules (170, 171, and 172) have been wrapped to work together and represent the possible behaviour of an AV alongside the Road Junction rules. Notice that this degree of abstraction (used in our approach) does not avoid the proper verification of the agent's behaviour to tell which rules have been selected by the agent. An example of such abstraction is noted in rule 172, where there are two different terms Give Way sign and triangle marked on the road with the same meaning. So, we only use the former term in our model and implementation.

3. Background

In this section we present some notation and concepts related to the models, languages and tools used in this work.

3.1. Timed Automata, Temporal Logic and UPPAAL

As presented in Baier and Katoen [27], timed automata model the behaviour of time-critical systems. A timed automaton has a finite set of clock variables. All clocks proceed at rate one. The value of a clock denotes the amount of time that has elapsed since its last reset. Conditions which depend on clock values are called clock (or time) constraints.

Definition 1 (Clock constraint)**.** *A clock constraint over a set C of clocks is formed according to the grammar g:*

$$g ::= x < c \mid x \leq c \mid x > c \mid x \geq c \mid g \wedge g$$

where $c \in \mathbb{N}$ and $x \in C$. $CC(C)$ represents the set of clock constraints over C.

The Timed Automaton definition [27] is given below.

Definition 2 (Timed Automaton)**.** *A timed automaton is a tuple $TA = (Loc, Act, C, \hookrightarrow, Loc_0, Inv, AP, L)$ where*

- *Loc is a finite set of locations,*
- *$Loc_0 \subseteq Loc$ is a set of initial locations,*
- *Act is a finite set of actions,*
- *C is a finite set of clocks,*

- $\hookrightarrow \subseteq Loc \times CC(C) \times Act \times 2^C \times Loc$ *is a transition relation,*
- $Inv{:}Loc \rightarrow CC(C)$ *is an invariant-assignment function,*
- AP *is a finite set of atomic propositions, and*
- $L{:}Loc \rightarrow 2^{AP}$ *is a labelling function for the locations.*

$ACC(TA)$ *denotes the set of atomic clock constraints that occur in either a guard or a location invariant of TA.*

Baier and Katoen [27] mention that *Timed Computation Tree Logic* (TCTL) is a real-time variant of temporal logic used to express properties of timed automata. So, the UPPAAL Model Checker which makes use of timed automata also uses a simplified version of TCTL to specify verification properties. Below, the syntax of TCTL is given (as seen in [27]) and also the corresponding syntax used in UPPAAL is provided in Table 1.

Definition 3 (Timed CTL syntax)**.** *Formulae in TCTL are either state or path formulae. TCTL state formulae over set AP of atomic propositions and set C of clocks are formed according to the* Φ *grammar:*

$$\Phi ::= true \mid a \mid g \mid \Phi \wedge \Phi \mid \neg\Phi \mid \exists\varphi \mid \forall\varphi$$

where $a \in AP$, $g \in ACC(C)$ *and* φ *is a path formula defined by:*

$$\varphi ::= \Phi \bigcup^{J} \Phi$$

where $J \subseteq \mathbb{R}_{\geq 0}$ *is an interval whose bounds are natural numbers.*

NB: the propositional logic operators $(\vee, \rightarrow, etc)$ are obtained from $\vee$ and $\neg$. Also, the temporal logic operators $\Box$ and $\Diamond$ are obtained by using existing operators in Φ and φ grammars.

In Table 1, we show the UPPAAL syntax (based on TCTL) used to write formulae and temporal properties.

Table 1. UPPAAL syntax.

Operator	Meaning
&&	And
||	Or
==	Equivalence
imply	Conditional
not	Negation
A	Universal quantifier
E	Existential quantifier
[]	Always
<>	Eventually
-->	Leads to

Below, we show an example of a formula written using UPPAAL syntax.

```
A[] (AV.at_roadjunction imply AV.enter_roadjunction)
```

This formula states: for all possible paths it is always the case that if the AV is placed at the road junction it will enter the road junction.

NB: The above example could be slightly changed to: $\forall\Box($`AV.at_roadjunction`$\rightarrow \forall\Diamond$`AV.enter_roadjunction`$)$, using TCTL notation. However, UPPAAL does not allow nesting of path formulae in a way that to write this formula, it is necessary to use the operator leads to ($\rightsquigarrow$). The previous TCTL formula may expressed using UPPAAL syntax as: `AV.at_roadjunction --> AV.enter_roadjunction`.

3.2. BDI Model and GWENDOLEN Language

In our work we use the GWENDOLEN agent programming language [15] in order to implement a BDI agent [28] to capture the core decision-making behaviour of an autonomous vehicle. By using GWENDOLEN, we can also take advantage of the MCAPL framework [17], where the AJPF model checker can be used to formally verify the behaviour of the agent. The MCAPL framework allows us to program BDI agents in languages such as GWENDOLEN and Goal [29], and one can also program the agent's environment using Java. In addition, it is possible to use the AJPF model checker to verify the agent's programming, where it is possible to check the agent's behaviour. AJPF is an extension of Java PathFinder (JPF) [30] which is, in turn, a tool for model-checking Java programs.

3.2.1. BDI Model

As described in Bordini et al. [31], the Beliefs-Desires-Intentions (BDI) Model is based on a model of human behaviour developed by Bratman [28].

- Beliefs are information the agent has about the world.
- Desires are all possible states of events that the agent might want to achieve.
- Intentions are the state of events the agent has decided to commit towards. These events can be goals that are assigned to the agent or the agent may choose among a set of options.

When implementing a BDI model in an agent programming language we usually have the following structure for an agent plan:

```
trigger_event : guard <- body
```

where a given agent may have different plans in order to achieve a certain goal.

- The `trigger_event` is given by a new belief or a goal.
- The `guard` is defined by a set of beliefs.
- The `body` is represented as a set of actions.

Example

We provide a simple example considering the AV-agent at a road junction.

```
AV-agent believes it is at the road junction.
AV-agent selects the intention to enter the road junction.
AV-agent triggers the following plan:
enter-roadjunction : at-roadjunction <- check-sign, watchout-for-road-user;
```

In this example initially the AV-agent believes it is placed at the road junction, next it has the desire to enter the road junction and selects an intention to achieve this goal. As a consequence it triggers a plan to execute two actions: the first one checks the existing traffic sign at the road junction and the second action is responsible for watching out for any road user crossing the junction.

3.2.2. GWENDOLEN Language

GWENDOLEN is an agent declarative logic-programming language incorporating explicit representations of goals, beliefs, and plans. The language uses similar syntactic conventions to other BDI agent languages. Here, we describe the syntax elements used in our implementation:

`+b` adds the belief `b`.
`-b` removes the belief `b`.
+!g adds the goal g.
+!g[perform] adds a new goal of type perform. Perform goals are discharged by the execution of an appropriate plan.

+!g[achieve] adds a new goal of type achieve. Achieve goals are discharged only when they become beliefs.
`B x` represents a guard condition, checks if belief `x` is perceivable.
`G x` represents a guard condition, checks if goal `x` has been added.
`hello(x)` represents that action `hello(x)` (defined in the agent's environment) is executed.

A plan in GWENDOLEN uses the syntax previously presented in a BDI model.

Example

We retake the previous example of the AV-agent, but now using GWENDOLEN syntax.

```
at(roadjunction)            \\ predefined belief ‘‘agent is at the road junction’’
enter-roadjunction[achieve] \\ a goal (of type achieve) to ‘‘enter the road junction’’
+!enter-roadjunction[achieve] : { B at(roadjunction) } <-
check-sign(A,B), watchout-for-road-user(C,D);
```

In the last lines (above) there is a GWENDOLEN plan that represents the following: when the agent recognises the trigger event (i.e., the achieve goal of entering the road junction), it checks the guard (i.e., the predefined belief which says the agent is at the road junction), and then the agent executes two actions: `check-sign(A,B)` and `watchout-for-road-user(C,D)`. These actions are implemented in the agent's environment. NB: the values A, B, C, D represent coordinates in a grid which represents a road junction environment.

3.3. The Property Specification Language

The MCAPL framework provides a Property Specification Language (PSL) used to write properties for the AJPF Model Checker. In Table 2, we present the set of operators from PSL which is used in the verification of properties.

Table 2. PSL operators.

Operator	Meaning
<>	temporal logic operator Eventually
[]	temporal logic operator Always
B	a Belief of the agent
G	a Goal of the agent
I	an Intention of the agent
D	an Action of the agent
ItD	an Intention to execute an action of the agent
P	a Perception from the environment
&	logical operator And
||	logical operator Or
-->	logical operator Implies

Example

Using the same elements from the two previous examples, we can write a PSL specification.

```
<>((B(AV-agent,at(roadjunction))) & D(AV-agent,watchout-for-roaduser(1,0)))
```

The description of this specification is: eventually the AV-agent believes it is at the road junction and the AV-agent executes the action `watchout-for-roaduser` at postion $(1,0)$ (in the grid environment).

4. Modelling Using Timed Automata

The modelling of our system was carried out using timed automata within UPPAAL model checker tool. We have divided our model into two main automata: `AV_agent` and

`RJ_Env` (*Road Junction environment automaton*); and three additional (simple) automata which model specific artefacts from the road junction environment: `road_user` (*Watch out Road User automaton*), `safe_gap` (*Check for a Safe Gap automaton*), and `sign` (*Check traffic sign automaton*). These five automata set up a network of automata, which can all communicate with each other through synchronized channels. In our model, `RJ_Env` forms the main communication hub among all automata, receiving information from the artefacts as well as sending information to, and receiving information from, the `AV_agent`.

4.1. `AV_agent` Automaton

Figure 2 presents the automaton which models the basic behaviour of the AV-agent. The agent starts in a state where it is away from the RJ, the agent uses the communication channel to tell the `RJ_Env`automaton that it is going to approach the RJ. Once the agent is at the RJ, it will receive from the `RJ_Env` one of two possible alternatives that may exist at the RJ: (i) there is only the stop sign; or (ii) there are both the stop and the give way signs. At this moment, the clock (`x`) starts to work and the agent is at the state of watching out for RU. From this state, there are two possible outcomes: (i) `RJ_Env` tells the agent that RJ is free; or (ii) `RJ_Env` tells the agent RJ is busy. When the latter occurs the agent is supposed to start to wait until it is possible to watch again for road users. When the RJ is free, the agent checks for a safe gap and again two outcomes are possible: (i) there is a safe gap and the `RJ_Env` tells the agent to enter the RJ; or (ii) there is no safe gap, the agent should wait, so it moves to the waiting state (the same one which is used when the RJ is busy).

After this, the AV-agent has successfully entered the RJ, and it tells the `RJ_Env` that it is now away from the RJ once more.

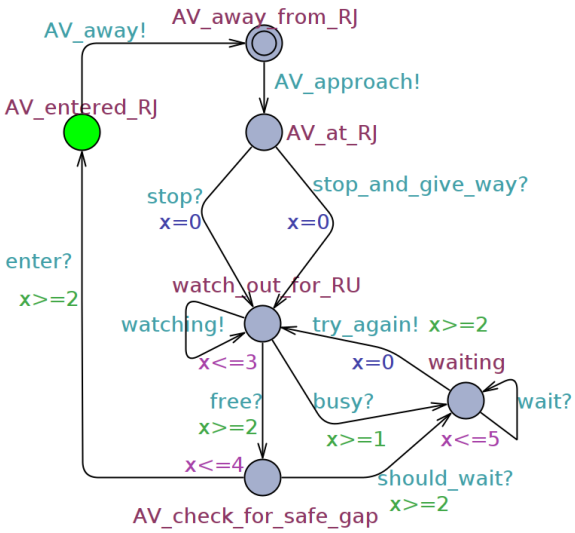

Figure 2. UPPAAL template for `AV_agent` automaton.

Time Constraints

To properly represent an AV, we have decided to add some time constraints to simulate thresholds for each one of the main actions in the system, i.e., watching for road users, waiting (at the RJ), check for a safe gap and entering the RJ.

Figure 3 illustrates how these time constraints work for the corresponding synchronized channels: busy, free, enter, should_wait, and try_again. The time constraints establish the lower and upper bounds for each one of the synchronized channels.

The lower and upper bounds for these time constraints have been selected considering that we could have a cross or a T junction, where the AV-agent should watch for road users at least in two different directions (in case of T junction) or at most in three directions (in case of a cross junction). However, in case the road junction (either cross or T junction) is busy, the AV-agent may only look once for road users and already check the RJ is busy.

Thus, the lower bound for the busy channel is 1 (one). For the remaining channels (enter, should_wait, and try_again) we only need to add one or two extra time units. The idea behind these extra time units is to model the additional time required for the AV-agent to execute its actions. For example, once the AV-agent checks the junction is free, then it will need an extra time unit to actually move and enter the junction.

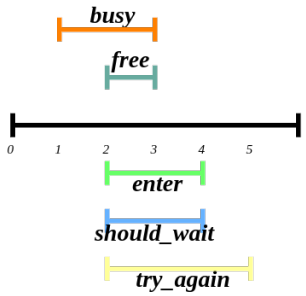

Figure 3. Time constraints for the `AV_agent` automaton synchronized channels.

4.2. Road Junction Environment Automaton

In Figure 4, the Road Junction environment automaton (`RJ_Env`) is shown. This model represents the behaviour and communication that the environment engages with the `AV_agent` and also with the three artefacts.

It starts in an idle state, as soon as the `AV_agent` approaches the RJ, the environment should check for the traffic sign, according to the information received from the `sign` artefact, the `RJ_Env` will tell the `AV_agent` if there is a single stop sign or two signs (stop and give way). After that, the `RJ_Env` waits for the `AV_agent` to start watching the RJ for road users. Now, two possible outcomes may be received from the `road_user` artefact: (i) the RU is away from the RJ; or (ii) the RU is crossing the RJ. When the latter occurs, the environment will notify the `AV_agent` that the RJ is busy, therefore the `AV_agent` is supposed to wait. Next, the `RJ_Env` waits to receive from the `AV_agent` the communication stating that it wants to try again and restart the checking for RU. But, in case the RJ is free, the environment will check for a safe gap with the corresponding `safe_gap` artefact. This artefact will answer whether or not there is a safe gap at the RJ. If there is no safe gap, the `RJ_Env` tells the `AV_agent` that it should wait at RJ. But, if there is a safe gap, thus the `RJ_Env` tells the `AV_agent` to enter the RJ. Finally, when the `AV_agent` tells the `RJ_Env` that it is away from the RJ, the environment is back to the idle state.

Notice that we use a variable `ru` (stands for Road User), which is incremented every time it perceives there is a Road User crossing the junction and is decreased every time there is a Road User away from the junction. So, before the synchronisation channel with the `AV_agent` is set up to communicate that the junction is free, it is necessary to check if `ru` is equal to zero (i.e., there is no Road User at all). If `ru` is not zero, there is still some Road User at the junction and the model does not proceed to the next stage, which is to check for a safe gap. NB: in the stage of verifying properties (see Section 6.1) we run simulations with one, two, and three Road Users, that is why we need this control variable `ru`.

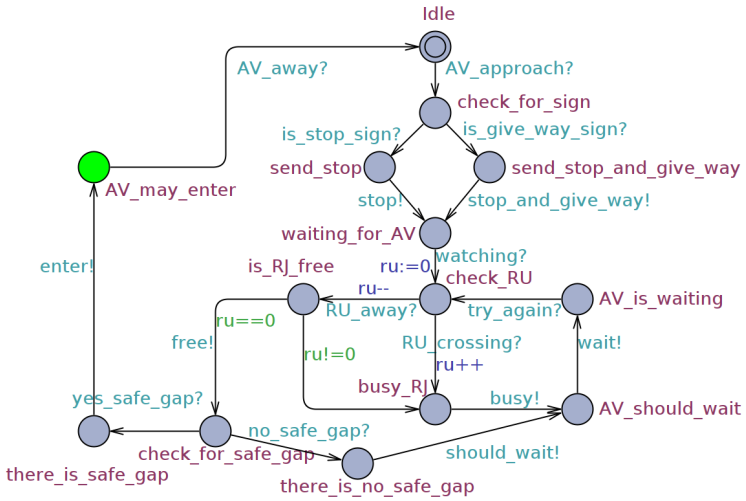

Figure 4. UPPAAL template for `RJ_Env` automaton.

4.3. Automata for the Artefacts

Figure 5 presents the three UPPAAL automata responsible for representing the behaviour of the artefacts from the environment (`RJ_Env`). The leftmost automaton shows the `sign` artefact, which should tell the environment the existing traffic sign (only stop sign or a stop and a give way signs). The centre automaton presents the `road_user` artefact, this one will send to `RJ_Env` the road user state (it is away from RJ or it is crossing). The rightmost automaton pictures the `safe_gap` artefact, which is responsible for telling whether or not there is a safe gap.

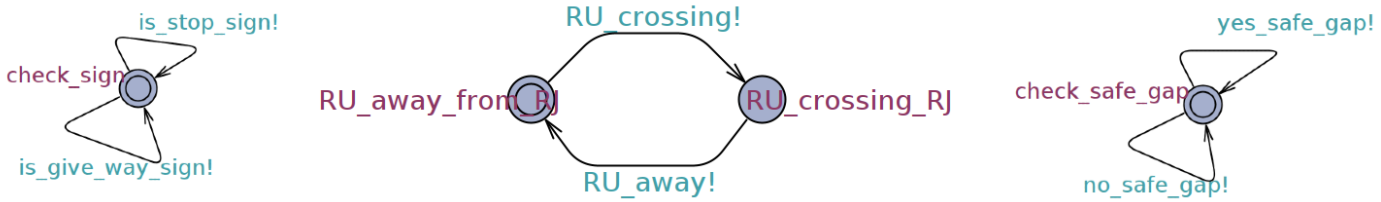

Figure 5. UPPAAL templates for the environment artefacts.

5. Agent and Environment Implementation

Our SAE-RoR system was implemented using the GWENDOLEN agent programming language and MCAPL framework. As previously shown in Figure 1, the implementation is based on LTL formalisation of RJ rules together with a mapping from the UPPAAL timed automata to the agent's implementation (this is presented later in Section 5.2). Nevertheless, some modifications were necessary because of the differences between UPPAAL and MCAPL frameworks. For example, by using UPPAAL we have modelled time constraints to represent the behaviour of the AV-agent in the road junction, while in MCAPL we have used random generation of events in the environment. In the following, we also describe the RJ environment modelling, implementation, and testing scenarios.

5.1. Setting-Up the Road Junction Environment Model

The model implemented using MCAPL is a simple representation of a crossroad junction. Figure 6 shows the grid which splits the road junction into nine spots. The grid set-up is as follows:

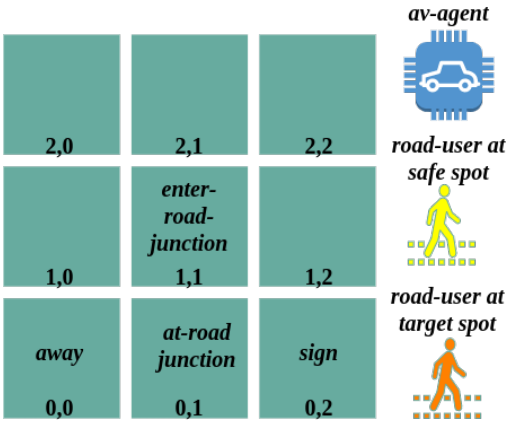

Figure 6. Road Junction grid environment model.

- spot `(0,0)`: the start position for the AV-agent, when it is said to be away from the road junction.
- spot `(0,1)`: the position the AV-agent goes when it is supposed to watch for road users.
- spot `(0,2)`: the position where the traffic sign is placed.
- spot `(1,1)`: the position reached by the AV-agent once it enters the RJ. Notice that after reaching `(1,1)` spot, the AV-agent may go to any of the following spots: {`(1,0)`; `(2,1)`; `(1,2)`}.
- spots {`(1,0)`); `(1,1)`); `(2,1)`; `(1,2)`} are said to be target spots. That is, spots that can be reached by the AV-agent.
- spots {`(0,0)`; `(2,0)`; `(2,2)`} are said to be safe spots. That is, spots that can not be reached by the AV-agent, once it arrives at the road junction, i.e., AV-agent is placed at `(0,1)`.
- a given road user may be placed at a safe or a target spot.

The above grid setup is implemented in the AV-agent as a set of initial beliefs.

5.2. Correspondence: Modelling and Implementation

Here we describe the correspondences from the UPPAAL modelling to the GWENDOLEN implementation. In Table 3, we describe the mapping from the `AV_agent` UPPAAL Automaton (previously seen in Figure 2) to the AV-agent implemented in GWENDOLEN (previously shown in Listing 1).

Notice that the information in the table is separated into *names* and *types* both for the UPPAAL model and the agent's implementation. For the model, the *names* represent the *Locations* or the *communication channels* used in the `AV_agent` Automaton, while the *types* identify which element this name represents, it can be a *Location* (**Loc**) or a communication channel with other UPPAAL Automata. In this case, we use the following representation:

UA1 is the `AV_agent` UPPAAL Automaton.
UA2 is the `RJ_Env` UPPAAL Automaton (see Figure 4).
UA3 is the `sign` UPPAAL Automaton (see Figure 5).
UA4 is the `road_user` UPPAAL Automaton (see Figure 5).
UA5 is the `safe_gap` UPPAAL Automaton (see Figure 5).

NB: when there are two pairs of different Automata as types, e.g., UA1-UA2/UA2-UA3 in the fourth row of the table. This means, the channel `stop?` is a communication from UA1 to UA2, while the channel `is_stop_sign?` synchronises the automata UA2 to UA3.

For the implementation, the *names* represent elements used in the GWENDOLEN code. These elements can be any of the following *types*:

Belief: represents an initial belief of the agent.

Add Belief: represents a new belief acquired by the agent during execution.
Percept: is a perception obtained by the agent from the environment.
Action: is an action executed by the agent in the environment.

Table 3. Correspondence between Model and Implementation.

Model		Implementation	
Names	**Types**	**Names**	**Types**
AV_away_from_RJ	Loc	av_away()	Belief
AV_approach!	UA1	approach_roadjunction()	Action
AV_at_RJ	Loc	at_roadjunction()	Percept
stop?/is_stop_sign?	UA1-UA2/U2-UA3	check_sign()/stop_sign()/stopped	Action/Percept/Add Belief
stop_and_give_way?/is_give_way_sign?	UA1-UA2/U2-UA3	check_sign()/give_way_sign()/ given_way	Action/Percept/Add Belief
watch_out_for_RU	Loc	watch()	Action
watching!	UA1	watching()	Action
busy?/RU_crossing?	UA1-UA2/UA2-UA4	road_user()/busy_roadjunction	Percept/Add Belief
wait?	UA2	wait/waiting(road_user)	Action/Percept
try_again!	UA1	try_again()	Percept
free?/RU_away?	UA1-UA2/UA2-UA4	no_road_user()/free_roadjunction	Percept/Add Belief
check_for_safe_gap	Loc	check_safe_gap()	Action
should_wait?/no_safe_gap?	UA1-UA2/UA2-UA5	no_safe_gap()/checking()	Percept/Action
enter?/yes_safe_gap?	UA1-UA2/UA2-UA5	safe_gap() or for_safe_gap()/enter	Percept/Action
AV_entered_RJ	Loc	enter_roadjunction	Percept
AV_away!	UA1	away_from_roadjunction	Add Belief

The mapping presented is direct where a given element from the Model has a matching element in the implementation. An additional correspondence, (that is not shown in the previous table) is the one from the UPPAAL Automata `sign`, `road_user`, and `safe_gap`, which are mapped to the random generation of these three events in the agent's environment.

However, not all elements can be mapped between the model and the implementation. For example, the `AV_agent` UPPAAL Automaton uses clock constraints that do not have a corresponding element in the agent's implementation. In addition, the GWENDOLEN code also has some details which are abstracted away in the timed model. The AV-agent has specific plans for different road junction rules (see the goals `enter_roadjunction_rules170_171` and `enter_roadjunction_rules170_172` in Listing 1). In this way, it is possible to keep track of which rules have been selected by the AV-agent.

Listing 1. AV-agent plans.

```
:Plans:

+!at_roadjunction(X,Y) [achieve] : { B av_away(0,0), B roadjunction(X,Y) }
    <- approach_roadjunction(X,Y);

+at_roadjunction(X,Y) : { B sign(Z,W) } <- check_sign(Z,W);

+stop_sign(Z,W) : { B sign(Z,W) }
    <- +stopped, +!enter_roadjunction_rules170_171[perform];

+give_way_sign(Z,W) : { B sign(Z,W) }
    <- +given_way, +stopped, +!enter_roadjunction_rules170_172[perform];

+!enter_roadjunction_rules170_171[perform] :
   { B at_roadjunction(X,Y), B stopped, B to_watch(S,T) }
     <- watch(S,T);

+!enter_roadjunction_rules170_172[perform] :
   { B at_roadjunction(X,Y), B given_way, B stopped, B to_watch(S,T) }
     <- watch(S,T);

+for_road_users(S,T) : { B road_user(S,T) }
    <- +busy_roadjunction, wait;
```

```
+waiting(road_user) : { B road_user(S,T) }
    <- watching(S,T);

+for_road_users(S,T) : { B no_road_user(S,T) }
    <- +free_roadjunction, check_safe_gap(S,T);

+try_again(S,T) : { B no_road_user(S,T) }
    <- +free_roadjunction, -busy_roadjunction, check_safe_gap(S,T);

+safe_gap(S,T) : { B no_road_user(S,T) }
    <- enter;

+no_safe_gap(S,T) : { B no_road_user(S,T) }
    <- checking(S,T);

+for_safe_gap(S,T) : { B new_safe_gap(S,T), B no_road_user(S,T) }
    <- enter;

+enter_roadjunction : { True }
    <- +away_from_roadjunction, done;
```

5.3. Implementation Details

Here, the implementation details concerning the AV-agent plans written in GWENDOLEN and the agent environment are described. Listing 1 presents a fragment of the agent implementation.

The first plan of the agent is designed to make the agent approach the RJ, so the action `approach_roadjunction(X,Y)` is invoked in the environment. This will only happen when the agent acquires the goal (of type achieve) `at_roadjunction(X,Y)` and has as guards the two beliefs: `av_away(0,0)` and `roadjunction(0,0)`. Next, the action `check_sign(Z,W)` is called, this action uses a random procedure to generate one of two possible outputs for traffic sign: stop or give way sign. To run this action the agent needs to perceive that it is `at_roadjunction(X,Y)` and believe there is a sign at (Z, W).

According to the existing traffic sign, a specific plan will be triggered. With this, we could track which RJ rule has been used by the agent. Either way, the agent will eventually call the action `watch(S,T)`, which is responsible for watching for road users. This action will return one of the two perceptions (from the environment): there is a road user or there is no road user.

In case there is a road user (i.e., exists a belief `road_user(S,T)`), the actions `wait` and `watching(S,T)` are executed. The former action will trigger a delay and the latter action is responsible for making the agent watch again for road users. The action `watching(S,T)` uses a random generation of road users at the road junction, in a way that at some point the road junction is supposed to be free of road users.

In case there is no road user (i.e., exists a belief `no_road_user(S,T)`), the agent believes the road junction is free and the action `check_safe_gap(S,T)` is executed. This action works similarly to action `check_sign(Z,W)` because it also uses a random generation to determine whether (or not) there is a safe gap at the road junction.

If there is no safe gap (i.e., exists a perception `no_safe_gap(S,T)`), a new action `checking(S,T)` is invoked, this action works similarly to action `watching(S,T)`, since it also uses random generator until at some point a safe gap arises at the road junction. Notice that to run action `checking(S,T)` the agent should still belief that there is no road user.

If there is a safe gap (and knowing that there is no road user), then the AV-agent may successfully enter the road junction. Once the agent has entered, it acquires a perception `enter_roadjunction`. After that, a new belief is added to the agent, so the agent knows that now it is away (once more) from the road junction (`away_from_roadjunction`).

5.4. Testing Scenarios

To test the SAE-RoR implementation stage we have run three different scenarios (see Figure 7). The setup of these scenarios corresponds to the placement of the road users in the road junction environment, which are the positions the AV-agent is supposed to watch for.

1. There are three road users, all at target spots, {(1,0); (1,1); (1,2)}.
2. There is one road user at a target spot, (1,0). And two road users at safe spots, {(2,0); (2,2)}.
3. There are three road users, all at safe spots, {(0,0); (2,0); (2,2)}.

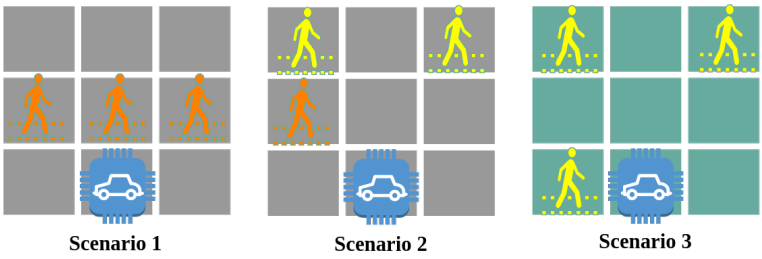

Figure 7. Three testing scenarios.

Figure 8 shows the output log from scenario 2. Notice that rule 171 is selected by the agent and action `watching(1,0)` is executed until the road junction is free, similarly, action `checking(1,0)` is also executed until there is a safe gap and the AV-agent may enter the road junction.

```
<terminated> run-AIL [Java Application] /usr/lib/jvm/java-8-openjdk-amd64/bin/java (Feb 23, 2021, 2:56:05 PM – 2:56:05 PM)
MCAPL Development Version 2019
ail.mas.DefaultEnvironment[INFO|main|2:56:05]: av done approach_roadjunction(0,1)
Stop sign (rule 171).
ail.mas.DefaultEnvironment[INFO|main|2:56:05]: av done check_sign(0,2)
AV has stopped
Busy at road user 1 position.
Free at road user 2 position.
Free at road user 3 position.
ail.mas.DefaultEnvironment[INFO|main|2:56:05]: av done watch(1,0)
ail.mas.DefaultEnvironment[INFO|main|2:56:05]: av done wait
keep watching until road junction is free
Road junction is free!
ail.mas.DefaultEnvironment[INFO|main|2:56:05]: av done watching(1,0)
AV has watched again
Is there a safe gap? false
ail.mas.DefaultEnvironment[INFO|main|2:56:05]: av done check_safe_gap(1,0)
No safe gap. Check again
Now, is there a (new) safe gap? true
ail.mas.DefaultEnvironment[INFO|main|2:56:05]: av done checking(1,0)
ail.mas.DefaultEnvironment[INFO|main|2:56:05]: av done enter
```

Figure 8. Scenario 2: output log.

As outlined above, the environment implementation has some actions that use random generation of events. As a result, we have run for each one of the three scenarios four different instances, this is necessary to properly capture all the possible outcomes of the actions. Specifically, we have observed the following elements:

- which RJ rule has been selected: rule 171 or 172.
- whether the RJ initially is busy or free.
- if initially the RJ has a safe gap or there is no safe gap.
- and whether or not the AV-agent has entered the RJ.

6. Formal Verification Results

In this section, we present the obtained results of our double-level model checking technique, where we have applied formal verification at design (using UPPAAL) and development (using MCAPL) levels.

All simulations and verifications (presented in this section) were done using the following specification: OS: *Linux Mint 19.3*; Processor: *Intel i7-8550U*; RAM: *8 GB*. And the correspondent software versions: UPPAAL *4.1.24—Academic*; MCAPL *development version*

2019 (NB: Our repository is available at https://github.com/laca-is/SAE-RoR, accessed on 23 June 2021).

We have successfully verified 30 properties (18 using UPPAAL and 12 using AJPF). These properties intend to capture all possible scenarios *w.r.t* the agent's behaviour against the three road junction rules (rules 170, 171, 172). With this, we intend to verify whether or not the agent is respecting the traffic rules according to the existing artefacts in the road junction environment.

6.1. Verification of Properties with UPPAAL

Below, we list the 18 properties written in TCTL that were successfully verified using UPPAAL. NB: `AV.x` represents the clock used in time constraints for the `AV_agent` automaton.

```
p1: A[] not deadlock
```

Description: a safety property which verifies if there is no deadlock.

```
p2: A[] ((RoadJunction.send_stop || RoadJunction.send_stop_and_give_way)
imply AV.AV_at_RJ)
```

Description: For all paths always the Road Junction environment when sending the `AV` to stop or to stop and give way to traffic, then the `AV` will be at the Road Junction.

```
p3: A[] (RoadJunction.waiting_for_AV imply A<> AV.watch_out_for_RU)
```

Description: For all paths always when the Road Junction is waiting for the `AV`, then for all paths at some time the `AV` watches out for road users.
NB: for the sake of clarity we use TCTL notation for this property, see in Section 3.1 the corresponding UPPAAL notation.

```
p4: A[] (AV.AV_check_for_safe_gap imply (RoadJunction.check_for_safe_gap ||
RoadJunction.there_is_safe_gap || RoadJunction.there_is_no_safe_gap))
```

Description: For all paths always when the `AV` checks for safe gap, then the Road Junction will be checking for a safe gap or it will know if (or not) there is a safe gap.

```
p5: A[] (RoadJunction.AV_may_enter imply A<> AV.AV_entered_RJ)
```

Description: For all paths always when the Road Junction tells the `AV` that it may enter the junction, then for all paths at some time the `AV` will enter the junction.
NB: the same remark for `p3` is valid for `p5`.

```
p6: A[] (AV.AV_entered_RJ imply AV.x >= 2)
```

Description: For all paths always when the `AV` enters the Road Junction the clock (x) has a value greater or equal than 2.

```
p7: A[] ((AV.watch_out_for_RU) imply (AV.x >= 0 && AV.x <= 3))
```

Description: For all paths always the when the `AV` watches for Road Users at the Road Junction the clock (x) has a value between 0 and 3.

```
p8: A[] ((AV.waiting) imply (AV.x >= 1 && AV.x <= 5))
```

Description: For all paths always when the `AV` waits at the Road Junction the clock (x) has a value between 1 and 5.

```
p9: A[] ((AV.AV_check_for_safe_gap) imply (AV.x >= 2 && AV.x <= 4))
```

Description: For all paths always the when the AV checks for a safe gap at the Road Junction the clock (x) has a value between 2 and 4.

```
p10: A[] ((AV.watch_out_for_RU) imply
(RoadUser1.RU_crossing_RJ || RoadUser1.RU_away_from_RJ))
```

Description: For all paths always when the AV watches for (a single) road user, then it is only possible to have the road user crossing or away from the junction.

```
p11: A[] (RoadUser1.RU_crossing_RJ imply (RoadJunction.is_RJ_free || RoadJunction.busy_RJ
|| RoadJunction.AV_should_wait || RoadJunction.AV_is_waiting || RoadJunction.check_RU))
```

Description: For all paths always when there is a (single) road user crossing the junction, then it is only possible the Road Junction (environment) is checking for a road user or it is waiting or it should wait or it knows the junction is busy or yet it should check if the junction is free.

```
p12: A[] (RoadUser1.RU_crossing_RJ imply (not AV.AV_entered_RJ))
```

Description: For all paths always when there is a (single) road user crossing the junction, then it is not possible that the AV will enter the junction.

```
p13: A[] ((AV.watch_out_for_RU) imply ((RoadUser1.RU_crossing_RJ ||
RoadUser1.RU_away_from_RJ) || (RoadUser2.RU_crossing_RJ || RoadUser2.RU_away_from_RJ)))
```

Description: this is basically the same property as p10, except that here there are two Road Users at the Road Junction.

```
p14: A[] ((RoadUser1.RU_crossing_RJ || RoadUser2.RU_crossing_RJ) imply
(RoadJunction.is_RJ_free || RoadJunction.busy_RJ || RoadJunction.AV_should_wait ||
RoadJunction.AV_is_waiting || RoadJunction.check_RU))
```

Description: this is basically the same property as p11, except that here there are two Road Users at the Road Junction.

```
p15: A[] ((RoadUser1.RU_crossing_RJ || RoadUser2.RU_crossing_RJ) imply
(not AV.AV_entered_RJ))
```

Description: this is basically the same property as p12, except that here there are two Road Users at the Road Junction.

```
p16: A[] ((AV.watch_out_for_RU) imply ((RoadUser1.RU_crossing_RJ ||
RoadUser1.RU_away_from_RJ) || (RoadUser2.RU_crossing_RJ || RoadUser2.RU_away_from_RJ)
|| (RoadUser3.RU_crossing_RJ || RoadUser3.RU_away_from_RJ)))
```

Description: this is basically the same property as p10 and p13, except that here there are three Road Users at the Road Junction.

```
p17: A[] ((RoadUser1.RU_crossing_RJ || RoadUser2.RU_crossing_RJ ||
RoadUser3.RU_crossing_RJ) imply (RoadJunction.is_RJ_free || RoadJunction.busy_RJ ||
RoadJunction.AV_should_wait || RoadJunction.AV_is_waiting || RoadJunction.check_RU))
```

Description: this is basically the same property as p11 and p14, except that here there are three Road Users at the Road Junction.

```
p18: A[] ((RoadUser1.RU_crossing_RJ || RoadUser2.RU_crossing_RJ ||
RoadUser3.RU_crossing_RJ) imply (not AV.AV_entered_RJ))
```

Description: this is basically the same property as p12 and p15, except that here there are three Road Users at the Road Junction.

In Table 4 the execution results from the properties are summarised considering the existence (or not) of road users in the scenarios as well as the time and memory used to run each set of properties. Notice the highest values for time and memory respectively are 0.01 s and 49,396 KB, which can be seen as fair values even when three road users are considered in the simulation.

Table 4. Properties verified with UPPAAL—Execution results.

Properties	Scenario	Time	Memory
p1–p12	with 0 or 1 RU	0 s to 0.003 s	5800 KB/49,396 KB
p13–p15	with 2 RU	0.001 s	6040 KB/48,322 KB
p16–p18	with 3 RU	0.001 s to 0.01 s	6040 KB/48,322 KB

Table 5 presents the results from the 18 properties checked using UPPAAL. In this table we have classified each property according to the following:

- Road users: no road user at all; one, two, or three road users.
- System properties: two kinds of system properties are considered: temporal correctness and liveness.
- Interaction: that is those properties that present some sort of interaction with the environment.
- Quality: there are two kinds of properties related to quality: security and safety.
- Related Road Junction rules: each property is identified with the correspondent Road Junction rules that are related to the verified property.

Table 5. Properties verified with UPPAAL—Classification.

		System Properties		Interaction	Quality		Related Rules		
Property #	**Road Users?**	**Temporal Correctness**	**Liveness**	**Interaction w/Environment**	**Safety**	**Security**	**R. 170**	**R. 171**	**R.172**
p1	-					•			
p2	-			•	•			•	•
p3	-		•	•			•		
p4	-			•	•		•	•	
p5	-		•	•			•		
p6	-	•					•		
p7	-	•					•		
p8	-	•					•		
p9	-	•					•	•	
p10	1			•	•		•		
p11	1			•	•		•		
p12	1			•	•		•		
p13	2			•	•		•		
p14	2			•	•		•		
p15	2			•	•		•		
p16	3			•	•		•		
p17	3			•	•		•		
p18	3			•	•		•		

To discuss the verified properties and results we highlight some issues, as follows.

1. Properties `p1` to `p5` are related to security, safety, liveness, and interaction. In addition, these properties verify some of the main actions of our model, i.e., when AV-agent watches out for a road user, checks for a safe gap and for a traffic sign as well as will enter the RJ.

2. Properties p6 to p9 are responsible for verifying the time constraints included in the AV-agent automaton. With this, we can check temporal correctness for the main actions in our model, i.e., enter the RJ, watch out for road users, wait at RJ, and check for a safe gap.
3. Properties p10 to p12 are safety properties used to verify the effect of having a single road user at the RJ in some related actions. These properties formally verify what to expect when the AV-agent watches out road users and also when there is a road user crossing the junction what is allowed (and not) to happen considering the existent actions in the RJ environment.
4. Properties p13 to p15 run the same kind of verification from the previous item, except here the scenario considers the existence of two road users.
5. Properties p16 to p18 run the same kind of verification from item 3, except here the scenario considers the existence of three road users.
6. Related RJ rules: 16 properties are related to rule 170, which is indeed a general road traffic rule handling different possibilities of when and how a vehicle may enter the road junction. Moreover, rules 171 and 172 are also verified in specific properties.

The verification of properties with UPPAAL generates important information for stakeholders. Firstly, it is possible to check the main actions that can be taken by an AV-agent at Road Junction. Secondly, the time constraints included in our model which, as noted, are left implicit in non-digital highway codes, were shown to be reasonable and so can form recommendations for a Digital Highway Code. Thirdly, the model is efficient at analysing the scenario with three road users, where there is no increase in the use of time and memory. As a result, we believe it would be feasible to analyse more complex road junction models with more than three road users. Lastly, we have assessed the use of three Road Junction rules from the RoR, where the main actions and artefacts of each rule have been modelled and formally verified.

6.2. Verification of Properties with AJPF

We present the twelve properties (and their corresponding descriptions) that have been successfully verified with AJPF. NB: these properties are labelled with ap (representing *AJPF Property*) to distinguish them from the properties previously presented.

```
ap1: (B(av,sign(0,2)) & B(av,stopped)) -> [] G(av, enter_roadjunction_rules170_171)
```

Description: when AV believes there is a sign at $(0,2)$ and it has stopped, then it always obtains the goal of entering the road junction using rules 170–171.

```
ap2: (B(av,sign(0,2)) & B(av,given_way) & B(av,stopped)) ->
[] G(av, enter_roadjunction_rules170_172)
```

Description: when AV believes there is a sign at $(0,2)$, it has given way and stopped, then it always obtains the goal of entering the road junction using rules 170–172.

```
ap3: [] (B(av, at_roadjunction(1, 0)) -> <> (B(av, road_user(1, 0)) ||
B(av, no_road_user(1, 0)))
```

Description: It is always the case that if the AV is at a road junction at $(1,0)$, then eventually it will believe that either there is a road user at the junction at $(1,0)$ or there is not a road user at the junction at $(1,0)$.

```
ap4: [] (D(av,wait) -> (B(av,road_user(1,0)) & B(av,busy_roadjunction)))
```

Description: It is always the case that if the AV waits at the junction, then it believes there is a road user at $(1,0)$ and the road junction is busy.

```
ap5: [] ((B(av,no_road_user(1,0)) & B(av,free_roadjunction)) -> <> (B(av,no_safe_gap(1,0)
|| B(av,safe_gap(1,0)) || B(av,new_safe_gap(1,0)) || B(av,try_again(1,0))))
```

Description: It is always the case that when the AV believes there is no road user at $(1,0)$ and the road junction is free, then eventually the AV will acquire the belief there is no safe gap at $(1,0)$ or there is a safe gap (or a new safe gap) at $(1,0)$ or the belief it has tried again at $(1,0)$ (in the search for road users).

```
ap6: [] (D(av,check_safe_gap(1,0)) -> ~B(av,busy_roadjunction))
```

Description: It is always the case that if the AV checks for safe gap at $(1,0)$, then it should not believe there is a busy road junction.

```
ap7: [] (D(av,check_safe_gap(1,0)) -> ~B(av,road_user(1,0)))
```

Description: It is always the case that if the AV checks for safe gap at $(1,0)$, then it should not believe there is a road user at $(1,0)$.

```
ap8: [] (D(av,check_safe_gap(1,0)) ->
(B(av,no_road_user(1,0)) & B(av,free_roadjunction)))
```

Description: It is always the case that if the AV checks for safe gap at $(1,0)$, then it believes there is no road user at $(1,0)$ and the road junction is free.

```
ap9: [] (D(av,enter) -> ~B(av,busy_roadjunction))
```

Description: It is always the case that if the AV enters the junction, then it should not believe there is busy road junction.

```
ap10: [] (D(av,enter) -> ~B(av,road_user(1,0)))
```

Description: It is always the case that if the AV enters the junction, then it should not believe there is a road user at $(1,0)$.

```
ap11: [] (D(av,enter) -> ~B(av,try_again(1,0)))
```

Description: It is always the case that if the AV enters the junction, then it should not believe to try again (and watch for a road user) at $(1,0)$.

```
ap12: [] (D(av, enter) -> ( B(av, safe_gap(1,0)) || B(av, new_safe_gap(1,0))
& B(av, no_road_user(1, 0)))
```

Description: It is always the case that if the AV enters the junction, then it believes there is a safe gap at $(1,0)$ (or a new safe gap) and no road user at $(1,0)$.

Table 6 shows the results obtained when running the AJPF model checker. These results consider Scenario 2 (previously seen in Figure 7), where there are three road users at the RJ, one of them is at a target spot and two are at safe spots.

All properties can be classified as safety properties. Properties `ap1` and `ap2` are specifically used to verify the application of the RJ rules, rules 170–171 and rules 170–172.

The remainder of the properties (from `ap3` to `ap12`) are responsible for verifying that the AV-agent performs key actions involved in the rules at appropriate points: that is to watch for road users, wait, check for a safe gap and enter the road junction. In Figure 9 the execution log of `ap12` is shown. In the last lines from the execution log (just above the results section) we notice the AV-agent knows the road junction is free (i.e., there is no road user) and that there is a new safe gap in the junction. This figure also shows no errors detected, in the results section of the execution log. This means that this property has been successfully model checked.

Table 6. Properties verified with AJPF—Execution results.

Property #	Results	Elapsed Time	States	Search	Instructions	Max Memory (MB)	Loaded Code
ap1	no errors detected	00:00:08	new = 703, visited = 201, backtracked = 904, end = 8	MaxDepth = 131, constraints = 0	36040819	603	Classes = 367, methods = 5647
ap2	no errors detected	00:00:08	new = 703, visited = 201, backtracked = 904, end = 8	MaxDepth = 131, constraints = 0	38107153	899	Classes = 368, methods = 5668
ap3	no errors detected	00:00:10	new = 703, visited = 201, backtracked = 904, end = 8	MaxDepth = 131, constraints = 0	39758810	731	Classes = 365, methods = 5630
ap4	no errors detected	00:00:07	new = 703, visited = 201, backtracked = 904, end = 8	MaxDepth = 131, constraints = 0	33160957	598	Classes = 368, methods = 5669
ap5	no errors detected	00:00:09	new = 703, visited = 201, backtracked = 904, end = 8	MaxDepth = 131, constraints = 0	45156165	896	Classes = 366, methods = 5651
ap6	no errors detected	00:00:07	new = 703, visited = 201, backtracked = 904, end = 8	MaxDepth = 131, constraints = 0	30367275	601	Classes = 367, methods = 5648
ap7	no errors detected	00:00:07	new = 703, visited = 201, backtracked = 904, end = 8	MaxDepth = 131, constraints = 0	31561846	601	Classes = 367, methods = 5648
ap8	no errors detected	00:00:07	new = 703, visited = 201, backtracked = 904, end = 8	MaxDepth = 131, constraints = 0	34589802	598	Classes = 367, methods = 5648
ap9	no errors detected	00:00:06	new = 703, visited = 201, backtracked = 904, end = 8	MaxDepth = 131, constraints = 0	30047443	602	Classes = 364, methods = 5629
ap10	no errors detected	00:00:06	new = 703, visited = 201, backtracked = 904, end = 8	MaxDepth = 131, constraints = 0	31242014	601	Classes = 367, methods = 5648
ap11	no errors detected	00:00:07	new = 703, visited = 201, backtracked = 904, end = 8	MaxDepth = 131, constraints = 0	31145552	600	Classes = 367, methods = 5648
ap12	no errors detected	00:00:08	new = 703, visited = 201, backtracked = 904, end = 8	MaxDepth = 131, constraints = 0	37515112	605	Classes = 368, methods = 5669

```
Busy at road user 1 position.
Free at road user 2 position.
Free at road user 3 position.
keep watching until road junction is free
keep watching until road junction is free
Road junction is free!
AV has watched again
Is there a safe gap? true
Is there a safe gap? false
No safe gap. Check again
No safe gap. Check again
Now, is there a (new) safe gap? true
Now, is there a (new) safe gap? true
Road junction is free!
Road junction is free!

====================================================== results
no errors detected

====================================================== statistics
elapsed time:       00:00:07
states:             new=703,visited=201,backtracked=904,end=8
search:             maxDepth=131,constraints=0
choice generators:  thread=1 (signal=0,lock=1,sharedRef=0,threadApi=0,reschedule=0), data=702
heap:               new=427079,released=422157,maxLive=5137,gcCycles=904
instructions:       37493424
max memory:         605MB
loaded code:        classes=367,methods=5648
```

Figure 9. Execution log of ap12.

Considering the obtained results (seen in Table 6), we highlight the following: (i) all properties have beensuccessfully verified; ii properties took from 6 (ap9 and ap10) to 10 (ap3) seconds; (iii) the results related to the states, search space, and loaded code are basically the same for all properties; (iv) the number of instructions ranges from 30.367.275 (lowest value by ap6) to 45.156.165 (highest value by ap5); and (v) the amount of memory (in MB) ranges from 598 (ap4 and ap8) to 899 (ap2). The similarity of the results is a consequence of the fact that most of the computation effort in AJPF is related to the production of an automata that represents the implemented program [32] which is identical in all cases here.

The formal verification with AJPF acknowledges and offers some addition to the previous verifications (carried out with UPPAAL). Firstly, we successfully verify that the main actions the AV-agent can take at the RJ (watch, wait, check for a safe gap, and

enter) are indeed invoked by the agent. Secondly, some properties include actions (e.g., `check_safe_gap`) that are only invoked in some cases (represented by the use of random in the environment) and so we also verify that these actions are taken when needed. Thirdly, we observed fair results in time, memory and other features obtained by the AJPF execution. At last, the verification process produces traces (and if necessary counter-examples) which allow us to identify which rules and random actions have been (autonomously) selected by AV-agent in any given scenario. This can be helpful, for example, if a scenario we verify leads to an accident, allowing a stakeholder to check and traceback the actions taken by the agent that led to that outcome and so advise on whether the agent, or possibly the representation of the rules in a Digital Highway Code, need to be amended.

7. Related Work

Here we analyse related work on the following topics: an AV application scenario, the Rules of the Road, some kind of Formal Verification technique (mainly Model Checking), some specification logic and the use of agents. Most of the works described here have as a goal the formal verification of a model related to AV.

In ref. [33], Luckcuck et al. present a survey on formal specification and verification of autonomous robotic systems. A number of these [34–37] apply formal verification to AVs, but none relate to our particular question around the design of Digital Highway Code rules that are intended to conform to pre-existing "Rules of the Road".

Table 7 summarises a comparison among the related work that is presented in the remainder of this section. The first three works [10,11,38] present some sort of formalisation for the road traffic rules (just like our approach does). Some interesting elements from these works are, correspondingly, the codification of traffic rules [10]; the solution for conflicts in traffic rules using a deontic logic [11]; the use of a real traffic data-set [38]. However, neither approach uses an agent abstraction to represent an AV decision-making. Kamali et al. and Al-Nuaimi et al. [12,13] include the formal verification (using Model Checking techniques) of BDI agents. But, their AV application scenario is not related to the road traffic rules.

Besides, Table 7 outlines some specific information concerning:

- Amount of road traffic rules used: some works (including our approach) represents 3 rules, but none represents more than this.
- Formal Verification tools: Ref. [12] and our work are the only ones that use two verification techniques at two different levels: design and development, in the other works a single technique is applied.
- Verification of properties: most properties are related to safety issues, but some include conflicts and consistency checking. Moreover, ref. [12] verifies 12 properties, ref. [13] 7, and ref. [10] 5 properties, while in our approach 30 properties are verified.
- Formalisation: all works use some kind of formalisation, most use temporal or deontic logic.
- Simulation tools: References [11–13] present the use of some graphical tool for simulation, which contribute for testing the system. Our approach uses the UPPAAL graphical tool for simulation, but for the agent's simulation we only use a cli (*command line interface*) tool.

  NB: all works described in this table share the same goal of using a formalisation technique to represent an AV, where either the road traffic rules are formalised or some formal verification of agents is used.

Table 7. Related work comparison.

Work Reference	AV Application Scenario	Road Traffic Rules	Amount of Rules	Formal Verification Tools	Verified Properties (Type/Amount)	Formalisation (Logic)	Agent Programming	Simulation and Assessment
[10]	urban traffic/ lane changes	Yes. Overtaking rules (German)	3	Isabelle/HOL Theorem Prover	safe distance/5 (theorems)	LTL	No	Uses Isabelle's code generator to codify the rules in Standard ML.
[11]	urban traffic/ lane changes	Yes. Overtaking rules (Australia)	1 (rule 141)	Turnip (DDL reasoning tool)	exceptions and conflicts/not specified	DDL	No	Uses CARRS-Q driving simulator to generate experiment data; w/4 different scenarios; find legal and illegal driving behaviour; help of domain experts; conducted 24 experiments.
[38]	urban traffic/ lane changes	Yes. Safe distance between vehicles (Vienna convention)	1	No	safe distance/ not specified	algebraic equations	No	Uses real traffic data-set (NGSIM project, US Highway 101) on position, speed, acceleration, and lane of vehicles. Simulates safe and unsafe lane changes.
[12]	AV platooning	No	-	UPPAAL/ MCAPL/AJPF	safety and liveness/12	TCTL/LTL	Yes. BDI Agent.	Uses TORCS (car simulator) for environment simulation; a physical engine is implemented in MATLAB.
[13]	parking lot	No	-	MCMAS	stability and consistency/7	CTL	Yes. BDI Agent.	A graphical environment is created using ROS and Gazebo.
Our approach	urban traffic/ road junction	Yes. Road Junction rules (UK)	3 (rules 170/ 171/172)	UPPAAL/ MCAPL/AJPF	safety/30	TCTL/LTL	Yes. BDI Agent.	Uses UPPAAL and AJPF tools for simulation, w/3 different scenarios. Simulates the use of road junction rules by the AV.

7.1. Formal Verification of Agents

Kamali et al. [12] use same tools used in our work, UPPAAL, AJPF, and GWENDOLEN to model, implement and verify a vehicle platooning protocol. They use a mixed strategy that combines results from UPPAAL and AJPF, to deduce properties both of individual agents in the platoon and overall platoon behaviour. Our approach uses the two Model Checkers separately in order to verify properties of a single agent following proposed Digital Highway Code rules at different levels of abstraction.

In our architecture, the model developed in UPPAAL is used as a design template for the lower-level agent implementation. Thus, our model checking stages are loosely coupled, which is beneficial for modularity, allowing, for instance, the design level UPAAL model to be implemented in a different programming language.

Al-Nuaimi et al. [13] use Agent Model Checking to explore the behaviour of an AV in a parking lot. Their toolchain consists of the MCMAS model checker, Jason agent programming language, and CTL to verify temporal properties. The authors formally verify the AVs decisions. 12 rules are defined to verify planning, navigation, object detection and obstacle avoidance. ROS and Gazebo [39] are used to graphically simulate the application scenario. Again this work is targeted at the verification of proposed AV implementations from a safety perspective rather than in terms of the digitisation of rules of the road and verifying whether some agent can obey them.

7.2. The Formalisation of the Rules of the Road

Pek et al. [38] formalise the safety of lane change manoeuvres to avoid collisions. The authors use as reference the Vienna Convention on traffic rules to formalise a single rule on the safe distance.

Rizaldi et al. [10] formalise and codify part of the German Highway Code on the Overtaking traffic rules in LTL. They show how the LTL formalisation can be properly used to abstract concepts from the traffic rules and obtain unambiguous and precise specification for the rules. In addition, they formally verify the traffic rules using Isabelle/HOL theorem prover and also monitor an AV applying a given traffic rule, which has been previously formalised using LTL.

Bhuiyan et al. [11] assess driving behaviour against traffic rules, specifically the Overtaking rules from the Queensland Highway Code. Two types of rules are specified: overtaking to the left and the right. Moreover, they intend to deal with rules exceptions and conflicts in traffic rules (this is solved by setting priorities among the rules). Using DDL (Defeasible Deontic Logic) they assess the driving behaviour telling if the driver has permission or it is prohibited to apply a given rule for overtaking. The results basically show if the proposed methodology has recommended (or not) the proper behaviour for the driver (permission or prohibition). In addition, CARRS-Q, a driving simulator is used and 24 experiments are conducted in four different scenarios.

Our approache share the same goal: assessing AV behaviour against traffic rules (in our case, the road junction rules). However, we are using an agent-based implementation and verification, where it is also possible to tell when and how a given road junction has been selected and applied by the agent. In addition, our double-level model checking architecture results in the formal verification of 30 properties (18 at design and 12 at development level), which brings a comprehensible set of verification that ranges from time constraints properties (at design level) to specific actions that can (or can not) be taken by the AV in the road junction scenarios (at development level). To the best of our knowledge, [10,11,38], do not present this variety of abstraction levels in the properties they verify.

8. Conclusions

In Section 1 we have introduced three questions that we wanted to answer (i) Can these three selected road junction rules be used directly (i.e. as seen in the Highway code) by an AV? (ii) How to assess the AVs behaviour against the three road junction rules

considering simple Road Junction scenarios? and (iii) Are there any guidelines that can be given to enable the AV to work correctly with such Road Junction rules?

The first question is answered by the formalisation and modelling proposed in our work. The Road Junction rules were abstracted and formalised into a Digital Highway code to render them machine-readable. To do this, it was necessary to remove ambiguity and make the rules explicit to the computational system. In addition, some degree of abstraction was necessary to handle similar terms as a single one, for example the term `safe-gap`, used in our work, can be found described in different ways throughout the rules from the UK Highway code.

The second question is answered by the own use of the double-level Model Checking technique and adoption of the methodology of exploring scenarios via random events from [18]. This generation of events makes it possible to simulate different scenarios within one model and explore all possible behaviours of the model's environment. By using the SAE-RoR architecture we have formally verified 30 properties (18 at the design level and 12 at the development level), these properties include security, safety, liveness, and temporal correctness properties, among others. We have obtained fair results considering the resources used (i.e., memory, time, search space, etc) in the verification of the properties (where all of these properties have been successfully verified). By running the verification of properties in the road junction simulated environment, we are able to capture and assess the AVs behaviour considering all possible actions (e.g., watch out, wait, check for safe gap, enter, etc) that can be taken by the AV-agent according to the three implemented road junction rules. Note that, while we do not claim that the properties verified completely represent all the possibilities, we believe that verification stages such as these will be necessary for reliable and compliant AVs.

For the third question, clearly we need a principled way to represent road junction rules in a machine-readable format. As part of this we need to identify and reify implicit the time constraints that appear in human-readable rules of the road. Similarly, the use of a BDI agent programming languages and Program Model Checking helps generate traces of AV-agent behaviour and so identify when and how a given Road Junction rule was applied. This kind of information is potentially of use to stakeholders.

We return to the trade-off mentioned in Section 1. Can a Digital Highway Code can be created with few minor changes or are several adaptions are necessary? We can only give an answer considering the subset from Road Junction rules that we have implemented here. These three rules express their ideas in sufficient detail for formal and executable representation in an AV. However, the rules still need some adaptation. In future work, we intend to revisit this question and develop a more general answer.

Having established the SAE-RoR architecture and workflow, we could now add the remaining Road Junction rules from the UK Highway Code. These remaining rules are similar to those already implemented, the differences lie primarily in the artefacts and the perceptions generated in the environment. For example, to add the Road Junction rules 175 and 176, which deal with Traffic Lights, we would need to represent the traffic light as an artefact and the green, amber, and red light as perceptions. But, the actions `stop` at the red light and `follow` at the green light, for instance, would not differ that much from actions already implemented for the AV-agent, like `wait` and `enter`. This work would be needed for full implementation of an AV but will yield little further insight at the methodological level.

Of more interest would be to consider a different section of traffic rules from the UK Highway Code, for example, the Roundabout rules in order to add generality to the framework. Similarly we could consider the inclusion of a Highway Code from a different country. Of particular interest would be to investigate how an AV-agent would work when travelling between countries when it would need to switch to a different set of Rules of the Road. The agent paradigm also allows us to explore behaviour in environments where agents have different profiles. Our AV-agent is supposed to behave according to the Road Junction rules. But, what will happen if it interacts with agents that violate traffic rules and

can this be modelled and verified? This extension would potentially introduce the need for the implementation and verification of communication and cooperation algorithms. Following this idea where we would have a multi-agent system, we notice that some other aspects offer an interesting view on how to extend the SAE-RoR architecture to consider the implementation and verification of AVs protocols. For instance, the topics of distributed traffic control [40], vehicle-to-vehicle and vehicle-to-infrastructure communication [41], and also agent-based IoT (*Internet of Things*) applications [42], however at this moment these lie outside the issue considered here of adherence to "The Rules of the Road".

Moreover, we could improve our abstract model from the road junction rules by defining an extension of the Multi-lane Spatial Logic, as seen in [23]. With this logic, we could extend our representation in a way not to only capture the temporal aspects from the road junction rules, but also the spatial elements. Perhaps, a proper approach to represent a safe gap in an urban traffic environment, for instance.

Lastly, we aim to augment the SAE-RoR architecture with an Ethical Agent responsible for monitoring and verifying an agent's behaviour with respect to the Rules of the Road, as discussed in [43].

Author Contributions: Conceptualization, G.V.A., L.D. and M.F.; Formal analysis, G.V.A. and L.D.; Funding acquisition, M.F.; Investigation, G.V.A.; Supervision, L.D. and M.F.; Writing—original draft, G.V.A.; Writing—review & editing, G.V.A., L.D. and M.F. All authors have read and agreed to the published version of the manuscript.

Funding: This research was partially funded in the UK by EPSRC project EP/V026801 (the Trustworthy Autonomous Systems Node in Verifiability) and by the Royal Academy of Engineering, under the Chair in Emerging Technologies scheme. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

Informed Consent Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Avary, M.; Dawkins, T. *Safe Drive Initiative: Creating Safe Autonomous Vehicle Policy*; World Economic Forum: Geneva, Switzerland, 2020.
2. Prakken, H. On the problem of making autonomous vehicles conform to traffic law. *Artif. Intell. Law* **2017**, *25*, 341–363. [CrossRef]
3. Alves, G.V.; Dennis, L.; Fisher, M. Formalisation of the Rules of the Road for embedding into an Autonomous Vehicle Agent. In Proceedings of the International Workshop on Verification and Validation of Autonomous Systems, Oxford, UK, 18–19 July 2018; pp. 1–2.
4. Alves, G.V.; Dennis, L.; Fisher, M. Formalisation and Implementation of Road Junction Rules on an Autonomous Vehicle Modelled as an Agent. In *International Symposium on Formal Methods*; Lecture Notes in Computer Science; Sekerinski, E., Moreira, N., Oliveira, J.N., Ratiu, D., Guidotti, R., Farrell, M., Luckcuck, M., Marmsoler, D., Campos, J., Astarte, T., et al., Eds.; Springer International Publishing: Cham, Switzerland, 2020; pp. 217–232. [CrossRef]
5. Philipp, R.; Wittmann, D.; Knobel, C.; Weast, J.; Garbacik, N.; Schnetter, P. *Safety First for Automated Driving*; Daimler AG: Stuttgart, Germany, 2019.
6. Law Commission, U. *Automated Vehicles: Summary of the Analysis of Responses to Consultation Paper 2 on Passenger Services and Public Transport*; Law Commission: London, UK, 2020.
7. The British Standards Institution. *PAS 1882 Data Collection and Management for Automated Vehicle Trials*; The British Standards Institution: London, UK, 2020.
8. Waymo. *Safety Report*; Waymo LLC: Mountain View, CA, USA, 2020. Available online: https://waymo.com/safety (accessed on 23 June 2021).
9. Department for Transport. Using the Road (159 to 203)—The Highway Code— Guidance—GOV.UK. 2017. Available online: https://www.gov.uk/guidance/the-highway-code/using-the-road-159-to-203 (accessed on 23 June 2021).
10. Rizaldi, A.; Keinholz, J.; Huber, M.; Feldle, J.; Immler, F.; Althoff, M.; Hilgendorf, E.; Nipkow, T. Formalising and Monitoring Traffic Rules for Autonomous Vehicles in Isabelle/HOL. In Proceedings of the 13th International Conference on Integrated Formal Methods, Turin, Italy, 20–22 September 2017; pp. 50–66. [CrossRef]
11. Bhuiyan, H.; Governatori, G.; Rakotonirainy, A.; Bond, A.; Demmel, S.; Islam, M.B. *Traffic Rules Encoding Using Defeasible Deontic Logic*; IOS Press: Amsterdam, The Netherland, 2020. [CrossRef]

12. Kamali, M.; Dennis, L.A.; McAree, O.; Fisher, M.; Veres, S.M. Formal Verification of Autonomous Vehicle Platooning. *Sci. Comput. Program.* **2017**, *148*, 88–106. [CrossRef]
13. Al-Nuaimi, M.; Qu, H.; Veres, S.M. Computational Framework for Verifiable Decisions of Self-Driving Vehicles. In Proceedings of the 2018 IEEE Conference on Control Technology and Applications (CCTA), Copenhagen, Denmark, 21–24 August 2018; pp. 638–645. [CrossRef]
14. Bakar, N.A.; Selamat, A. Agent systems verification: Systematic literature review and mapping. *Appl. Intell.* **2018**, *48*, 1251–1274. [CrossRef]
15. Dennis, L.A. *Gwendolen Semantics: 2017*; Technical Report ULCS-17-001; Department of Computer Science, University of Liverpool: Liverpool, UK, 2017.
16. Bengtsson, J.; Larsen, K.; Larsson, F.; Pettersson, P.; Yi, W. UPPAAL—A tool suite for automatic verification of real-time systems. In *Hybrid Systems III*; Alur, R., Henzinger, T.A., Sontag, E.D., Eds.; Number 1066 in Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 1996; pp. 232–243. [CrossRef]
17. Dennis, L.A.; Fisher, M.; Webster, M.P.; Bordini, R.H. Model Checking Agent Programming Languages. *Autom. Softw. Eng.* **2012**, *19*, 5–63. [CrossRef]
18. Dennis, L.A.; Fisher, M.; Lincoln, N.K.; Lisitsa, A.; Veres, S.M. Practical Verification of Decision-Making in Agent-Based Autonomous Systems. *Autom. Softw. Eng.* **2016**, *23*, 305–359. [CrossRef]
19. Koeman, V.J.; Dennis, L.A.; Webster, M.; Fisher, M.; Hindriks, K.V. The "Why Did You Do That?" Button: Answering Why-Questions for End Users of Robotic Systems. *Lect. Notes Comput. Sci.* **2019**, *12058*, 152–172. [CrossRef]
20. Alves, G.V.; Dennis, L.; Fernandes, L.; Fisher, M. Reliable Decision-Making in Autonomous Vehicles. In *Validation and Verification of Automated Systems: Results of the ENABLE-S3 Project*; Leitner, A., Watzenig, D., Ibanez-Guzman, J., Eds.; Springer International Publishing: Cham, Switzerland, 2020; pp. 105–117. [CrossRef]
21. Fernandes, L.E.R.; Custodio, V.; Alves, G.V.; Fisher, M. A Rational Agent Controlling an Autonomous Vehicle: Implementation and Formal Verification. *Electron. Proc. Theor. Comput. Sci.* **2017**, *257*, 35–42. [CrossRef]
22. Dresner, K.; Stone, P. A Multiagent Approach to Autonomous Intersection Management. *J. Artif. Intell. Res.* **2008**, *31*, 591–656. [CrossRef]
23. Schwammberger, M. An abstract model for proving safety of autonomous urban traffic. *Theor. Comput. Sci.* **2018**, *744*, 143–169. [CrossRef]
24. Herrmann, A.; Brenner, W.; Stadler, R. *Autonomous Driving: How the Driverless Revolution Will Change the World*, 1st ed.; Emerald Publishing: Bingley, UK, 2018.
25. Nigeria, H.C. Nigeria Highway Code—III. ROAD JUNCTIONS. 2019. Available online: http://www.highwaycode.com.ng/iii-road-junctions.html (accessed on 23 June 2021).
26. Fisher, M. *An Introduction to Practical Formal Methods Using Temporal Logic*; Wiley: Hoboken, NJ, USA, 2011.
27. Baier, C.; Katoen, J.P. *Principles of Model Checking (Representation and Mind Series)*; The MIT Press: Cambridge, MA, USA, 2008.
28. Bratman, M.E. *Intentions, Plans, and Practical Reason*; Harvard University Press: Cambridge, MA, USA, 1987.
29. Hindriks, K.V. Programming Rational Agents in GOAL. In *Multi-Agent Programming: Languages, Tools and Applications*; El Fallah Seghrouchni, A., Dix, J., Dastani, M., Bordini, R.H., Eds.; Springer: Boston, MA, USA, 2009; pp. 119–157.
30. Visser, W.; Havelund, K.; Brat, G.; Park, S. Model Checking Programs. In Proceedings of the 15th IEEE International Conference Automated Software Engineering (ASE), Grenoble, France, 11–15 September 2000; pp. 3–12.
31. Bordini, R.H.; Hübner, J.F.; Wooldridge, M. *Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology)*; John Wiley & Sons, Inc.: Hoboken, NJ, USA, 2007.
32. Dennis, L.A.; Fisher, M.; Webster, M. Two-Stage Agent Program Verification. *J. Log. Comput.* **2018**, *28*, 499–523. [CrossRef]
33. Luckcuck, M.; Farrell, M.; Dennis, L.A.; Dixon, C.; Fisher, M. Formal Specification and Verification of Autonomous Robotic Systems: A Survey. *ACM Comput. Surv.* **2019**, *52*, 100:1–100:41. [CrossRef]
34. Althoff, M.; Althoff, D.; Wollherr, D.; Buss, M. Safety verification of autonomous vehicles for coordinated evasive maneuvers. In Proceedings of the 2010 IEEE Intelligent Vehicles Symposium, San Diego, CA, USA, 21–24 June 2010; pp. 1078–1083. [CrossRef]
35. Pallottino, L.; Scordio, V.G.; Bicchi, A.; Frazzoli, E. Decentralized Cooperative Policy for Conflict Resolution in Multivehicle Systems. *IEEE Trans. Robot.* **2007**, *23*, 1170–1183. [CrossRef]
36. Heß, D.; Althoff, M.; Sattel, T. Formal verification of maneuver automata for parameterized motion primitives. In Proceedings of the 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, Chicago, IL, USA, 14–18 September 2014; pp. 1474–1481. [CrossRef]
37. Kress-Gazit, H.; Wongpiromsarn, T.; Topcu, U. Correct, Reactive, High-Level Robot Control. *IEEE Robot. Autom. Mag.* **2011**, *18*, 65–74. [CrossRef]
38. Pek, C.; Zahn, P.; Althoff, M. Verifying the safety of lane change maneuvers of self-driving vehicles based on formalized traffic rules. In Proceedings of the 2017 IEEE Intelligent Vehicles Symposium (IV), Redondo Beach, CA, USA, 11–14 June 2017; pp. 1477–1483. [CrossRef]
39. Quigley, M.; Conley, K.; Gerkey, B.P.; Faust, J.; Foote, T.; Leibs, J.; Wheeler, R.; Ng, A.Y. ROS: An Open-source Robot Operating System. In Proceedings of the ICRA Workshop on Open Source Software, Kobe, Japan, 12–17 May 2009.
40. Bui, K.H.N.; Jung, J.J. Internet of agents framework for connected vehicles: A case study on distributed traffic control system. *J. Parallel Distrib. Comput.* **2018**, *116*, 89–95. [CrossRef]

41. Alouache, L.; Nguyen, N.; Aliouat, M.; Chelouah, R. Toward a hybrid SDN architecture for V2V communication in IoV environment. In Proceedings of the 2018 Fifth International Conference on Software Defined Systems (SDS), Barcelona, Spain, 23–26 April 2018; pp. 93–99. [CrossRef]
42. Savaglio, C.; Ganzha, M.; Paprzycki, M.; Bădică, C.; Ivanović, M.; Fortino, G. Agent-based Internet of Things: State-of-the-art and research challenges. *Future Gener. Comput. Syst.* **2020**, *102*, 1038–1053. [CrossRef]
43. Alves, G.V.; Dennis, L.; Fisher, M. First Steps towards an Ethical Agent for Checking Decision and Behaviour for an Autonomous Vehicle on the Rules of the Road. In Proceedings of the Second Workshop on Implementing Machine Ethics, Dublin, Ireland, 30 June 2020; Zenodo: Geneva, Switzerland, 2020; pp. 1–2. [CrossRef]