Science of Computer Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Formal verification of autonomous vehicle platooning

Maryam Kamali [a,*], Louise A. Dennis [a], Owen McAree [b], Michael Fisher [a], Sandor M. Veres [b]

[a] Department of Computer Science, University of Liverpool, UK
[b] Department of Automatic Control & Systems Engineering, University of Sheffield, UK

ARTICLE INFO

Article history:
Received 6 June 2016
Received in revised form 25 May 2017
Accepted 30 May 2017
Available online xxxx

Keywords:
Vehicle platooning
Agent programming
Model checking

ABSTRACT

The coordination of multiple autonomous vehicles into convoys or platoons is expected on our highways in the near future. However, before such platoons can be deployed, the behaviours of the vehicles in these platoons must be certified. This is non-trivial and goes beyond current certification requirements, for human-controlled vehicles, in that these vehicles can act *autonomously*. In this paper, we show how formal verification can contribute to the analysis of these new, and increasingly autonomous, systems. An appropriate overall representation for vehicle platooning is as a multi-agent system in which each agent captures the "autonomous decisions" carried out by each vehicle. In order to ensure that these autonomous decision-making agents in vehicle platoons never violate safety requirements, we use formal verification. However, as the formal verification technique used to verify the individual agent's code does not scale to the full system, and as the global system verification technique does not capture the essential verification of autonomous behaviour, we use a combination of the two approaches. This mixed strategy allows us to verify safety requirements not only of a model of the system, but of the actual agent code used to program the autonomous vehicles.

© 2017 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

While "driverless cars" regularly appear in the media, they are neither "driverless" nor fully autonomous. Legal constraints, such as the Vienna Convention [37], ensure that there must always be a responsible human in the vehicle. Although fully autonomous road vehicles remain futuristic, the automotive industry is working on what are variously called *road trains*, *car convoys*, or *vehicle platoons*. Here, each vehicle autonomously follows the one in front of it, with the lead vehicle in the platoon/convoy/train being controlled by a human driver. This technology is being introduced by the automotive industry in order to improve both the safety and efficiency of vehicles on very congested roads [29]. It is especially useful if the vehicles are trucks/lorries and if the road is a multi-lane highway.

In these platoons, each vehicle clearly needs to communicate with others, at least with the ones immediately in front and immediately behind. Vehicle-to-vehicle (V2V) communication is used at a lower (continuous control system) level to adjust each vehicle's position in the lanes and the spacing between the vehicles. V2V is also used at higher levels, for example to communicate joining requests, leaving requests, or commands dissolving the platoon. So a traditional approach is to implement the software for each vehicle in terms of hybrid (and hierarchical) control systems and to analyse this using hybrid systems techniques [4].

* Corresponding author.
E-mail address: maryam.kamali@liverpool.ac.uk (M. Kamali).

http://dx.doi.org/10.1016/j.scico.2017.05.006
0167-6423/© 2017 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

However, as these automotive platoons become more complex, there is a move towards much greater autonomy within each vehicle. Although the human in the vehicle is still responsible, the autonomous control deals with much of the complex negotiation to allow other vehicles to leave and join, etc. Safety certification is an inevitable concern in the development of more autonomous road vehicles, and verifying the safety and reliability of automotive platooning is currently one of the main challenges faced by the automotive industry. Traditional approaches to modelling such situations involve hybrid automata [18] in which the continuous aspects are encapsulated within discrete states, while discrete behaviours are expressed as transitions between these states. A drawback of the hybrid automaton approach is that it can be difficult to separate the two (high-level decision-making and continuous control) concerns. In addition, the representation of autonomous decision-making can become unnecessarily opaque in such hybrid approaches.

As is increasingly common within autonomous systems, we use a hybrid architecture where not only is the discrete decision-making component separated from the continuous control system, but the behaviour of the discrete part is described in much more detail; in particular, using the *agent* paradigm [40]. This style of architecture improves the system design from an engineering perspective and also facilitates system analysis and verification [9]. Indeed, we use this architecture for actually implementing automotive platoons, and we will here show how formal verification can be used for its direct analysis.

The verification of such systems is challenging due to their complex and hybrid nature. Separating discrete and continuous concerns, as above, potentially allows us to reason about the decision-making components in isolation and ensure that no decision-making component ever deliberately chooses an unsafe state. However, the use of the 'agent' concept alone is not enough for our purposes, since this can still make its autonomous decisions in an 'opaque' way. In order to be able to reason about, and formally verify, the choices the system makes, we use a *rational agent* [41]. This not only makes decisions, but has explicit representations of the *reasons* for making them, allowing us to describe not only what the autonomous system chooses to do, but *why* it makes its particular choices [16].

We utilise the *Belief-Desire-Intention* (BDI) model, one of the most widely used conceptual models, not only for describing these rational agents but for actually implementing them [32]. A BDI-style agent is characterised by its beliefs, desires and intentions: *beliefs* represent the agent's views about the world; *desires* represent the objectives to be accomplished; while *intentions* are the set of tasks currently undertaken by the agent to achieve its desires. A BDI-style agent has a set of plans, determining how an agent acts based on its beliefs and goals, and an event queue where events (perceptions from the environment and internal subgoals) are stored. There are several advantages in using this style of model for developing autonomous systems: (a) it naturally separates feedback controllers from high-level decision making, as above; (b) it facilitates reasoning and verifying about the behaviour of high-level decision making [11]; (c) it supports incremental and hierarchical development of plans; and (d) it provides a clear separation between plan selection and plan execution. However, a drawback of this form of approach is that it is essentially a plan management and plan selection framework with no in-built mechanisms for learning or first-principles planning. This means that a BDI agent does not automatically learn from past behaviour and adapt its plans accordingly. A similar limitation is lack of predictability and forward planning, in its basic form. As the aim of this paper is on *verification* of *decisions* concerning platooning, we can utilise this model in the form of the Gwendolen programming language [8], developed for verifiable BDI-style programming, to capture and implement the agent-based decision-making in each vehicle within an automotive platoon.

As part of safety verification, we need to verify the agent's decisions, especially in combination with the other vehicles. An autonomous rational agent makes decisions about what actions to perform, etc., based on the *beliefs*, *goals* and *intentions* that the agent holds at that time. We use a model-checking approach to demonstrate that the rational agent always behaves in line with the platoon requirements and never deliberately chooses options that end up in unsafe states. We verify properties of the rational agent code using the AJPF model-checker [12], one of the very few model-checkers able to cope with complex properties of BDI agents. Unfortunately, there are two drawbacks to using AJPF: currently, AJPF does not support verification of timed behaviours; and AJPF is resource heavy and cannot be used to verify the whole system. Consequently, we here propose a combined methodology for the verification of automotive platooning. To evaluate timing behaviour, we use a timed-automata abstraction and verify the system using Uppaal [2]; to evaluate individual autonomous decisions, we use AJPF together with an abstraction of the other vehicles/agents. Furthermore, we describe how these two approaches to modelling can be combined to provide an appropriate basis for verifying the behaviour of both individual agents and the whole system.

Overview of our contribution. The ISO 26262 standard provides a standard for *functional safety* management in automotive applications, and determines the safety requirements that should be fulfilled in design, development and validation of individual automotive units. Following the ISO 26262 guidelines for safety management in automotive applications we propose a verifiable agent-based architecture for development of safe automotive platooning and, in the same research line, we propose new combined verification techniques for autonomous systems developed based on hybrid agent architectures. In particular, we show the applicability of our verification techniques in the development of platooning.

For a clear picture of our approach, we summarise our fourfold contribution.

I We introduce formal *automotive platoons requirements*. This allows us to better understand the functioning of platooning protocols and, more importantly, to verify essential properties such as the functional correctness and liveness of the protocols. An important aspect of the protocols is that a vehicle can join and leave a platoon if, and only if, the whole platooning remains safe.

II We develop practical platooning protocols using a hybrid agent architecture where the high-level decision-making component of the platooning protocols is provided as Gwendolen agent code and the continuous control system is provided as a Simulink model. This contributes to *simulation and testing of protocols for validation*. The Simulink based control system was developed under the assumption that the physical model in the simulation is representative of real world vehicle dynamics. Exact correctness is not a requirement, as the controller has been shown to be robust to unmodelled dynamics [36,27].

III We propose *new combined verification techniques* for the safety analysis of systems which are developed using such hybrid agent architectures. This aims to separate different aspects of a system and the system properties, i.e., BDI and real-time requirements in order to empower the verification of complex autonomous systems.

IV We show how the proposed techniques for verification can be applied to *automotive platoon verification*. Our description of this includes a brief overview of the various types of models we use, in particular BDI agent models and Uppaal models, that are used for verification of the various system properties. Note that the verified BDI agent code is the actual code which has been used for simulation[1] and the real world testing.[2] This verification has been carried out under the assumption that the vehicle control and environment interactions are correct. This assumption is not restrictive because the vehicle control and environment interactions abstractly represent our Simulink based control system and, to ensure the correct functionality of the system control, an engineering analysis has been used [36,27]. The validity of our abstraction is checked by the simulation and testing, which is discussed in II and any errors found there lead to improved abstractions. Necessarily, real systems cannot be formally verified. Some abstractions *must* be used, though it is important to continually improve these abstractions.

Overall, the approach described here can be used to formally verify practical autonomous platooning systems.

The remainder of the paper is organised as follows. In Section 2 the automotive platoon and platoon requirements are presented. In Section 3 the hybrid agent architecture and the agent-based decision-making for the platoon are described. In Section 4 the analysis and verification of the platoon is considered. Finally, in Section 5, concluding remarks are provided.

2. Automotive platoons

An automotive platoon, enabling road vehicles to travel as a group, is led by a vehicle driven by a professional driver [35,39,34]. The following vehicles, i.e., members of the platoon, are controlled autonomously. These vehicles, equipped with low-level longitudinal (controlling speed) and lateral (controlling steering) control systems, travel in a platoon with predefined gaps between them. In addition, V2V communication also connects the vehicles. The lead vehicle effectively carries out coordination over the platoon: setting parameters, creating certificates for joining and leaving, etc. Each individual vehicle observes its environment and follows incoming commands from the lead vehicle. In what follows, we outline the set of high-level automotive platoon concepts and procedures including how to join and leave a platoon [3,29] in Section 2.1 and Section 2.2, respectively. In addition, the high-level requirements on these procedures for the development of safe and reliable platooning are explained. From these we derive the formal properties to be verified.

2.1. Joining the platoon

A vehicle can join a platoon either at the end or in the middle with different control strategies being used. The joining procedure is as follows:

- a non-member vehicle sends a joining request to the platoon leader, expressing the intended position in the platoon;
- if the vehicle has requested to join from the rear, the leader sends back an agreement provided the maximum platoon length has not been reached and the platoon is currently in normal operation, i.e., no other active joining/leaving procedures are happening;
- if the vehicle requests to join in front of vehicle X and the maximum platoon length has not been reached, the leader sends an “increase space” command to vehicle X, and when the leader is informed that enough spacing has been created, it sends back an agreement to the joining vehicle;
- upon receipt of an agreement, the joining vehicle changes its lane (changing lane is a manual procedure which is performed by a human driver);
- once the vehicle is in the correct lane, its automatic speed controller is enabled and it approaches the preceding vehicle;
- when the vehicle is close enough to the preceding vehicle, its automatic steering controller is enabled and it sends an acknowledgement to the leader; and, finally,
- the leader sends a “decrease space” command to vehicle X, and when the leader is informed that spacing has returned to normal, it replies to the acknowledgement.

[1] Platooning Simulation Demo http://wordpress.csc.liv.ac.uk/va/2016/05/18/autonomous-vehicle-platooning-demo/.
[2] Real World Testing Demo http://wordpress.csc.liv.ac.uk/va/2017/01/18/hw-testing-platooning/.

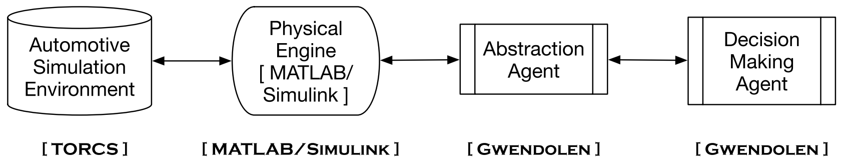


Fig. 1. Hybrid agent architecture.

In order to ensure safe joining operations, the following requirements should be preserved within the decision-making components of automotive platoon.

1. A vehicle must only initiate joining a platoon, i.e. changing lane, once it has received confirmation from the leader.
2. Before autonomous control is enabled, a joining vehicle must approach the preceding vehicle, in the correct lane.
3. Automatic steering controller must only be enabled once the joining vehicle is sufficiently close to the preceding vehicle.

2.2. Leaving the platoon

A vehicle can request to leave platoon at any time, i.e., a driver initiates a leaving request. The leaving procedure is:

- a platoon member sends a leaving request to the leader and waits for authorisation;
- upon receipt of 'leave' authorisation, the vehicle increases its space from the preceding vehicle;
- when maximum spacing has been achieved, the vehicle switches both its speed and steering controller to 'manual' and changes its lane; and, finally
- the vehicle sends an acknowledgement to the leader.

Note that the detailed communications with the driver are abstracted away. It means that the driver should be notified about the speed and steering controllers that are switched to 'manual'. As suggested in the EU legislation COMPANYON [29], a control and tell-tale device would be needed to let the driver take the vehicle control back and to inform the driver if (s)he is in the platoon. The two following requirements are also necessary properties for ensuring safe leaving operations that should be met with the agent-based decision-making components of an automotive platoon.

1. Except in emergency cases, a vehicle must not leave the platoon without authorisation from the leader.
2. When authorised to leave, autonomous control should not be disabled until the maximum allowable platoon spacing has been achieved.

Both leaving and joining procedures temporarily change the spacing of the platoon in different parts that could endanger safety of the platoon. There could be cases where the distance of two parts of the platoon would exceed the maximum permitted distance or a crash could be possible. An example scenario of such cases can be as follows: when a vehicle increases the distance to that preceding it to leave the platoon and at the same time the vehicle right behind the leaving vehicle increases its distance to let another vehicle to join to the platoon. If the leaving vehicle leaves the platoon before the joining vehicle has joined, the distance between two parts would be very large or a crash with the joining vehicle would be plausible if the platoon tries to decrease this distance. The platoon leader blocks the occurrence of both leaving and joining at the same time.

In the rest of the paper, we first explain our agent-based development of an automotive platoon. The development is based on the procedures introduced in Section 2. We then verify the correctness of joining and leaving procedures and in general the safety of the whole platoon.

3. Agent-based development of automotive platoon

We employ a hybrid agent architecture based on [10] for each vehicle (see Fig. 1).

Real-time continuous control of the vehicle is managed by feedback controllers, implemented in MATLAB. The real-time continuous control of the vehicle also observes the environment through its sensory input. This is called the *Physical Engine*. The Physical Engine, in turn, communicates with an *Abstraction Agent* that extracts discrete information from streams of continuous data and passes this on to a *Decision-Making Agent*. The Decision-Making Agent is a rational agent which directs the Physical Engine by passing it instructions through the Abstraction Agent. Instructions from the Decision-Making Agent to the Abstraction Agent are interpreted into meaningful instructions for the Physical Engine.

To provide the complex environment necessary for effective simulation and testing, we use an automotive simulator, TORCS,[3] to implement the environment component of the architecture. The Physical Engine is implemented in MATLAB,

[3] TORCS – The Open Racing Car Simulator https://sourceforge.net/projects/torcs/.

perform goal	+!g [perform]	adding a new goal to perform some actions
achievement goal	+!g [achieve]	adding a new goal and continuously attempting the plans associated with the goal until the agent has acquired the belief *g*
believe	+b	adding a new belief *b*
disbelieve	-b	removing a belief *b*
on-hold believe	*b	waiting for belief *b* to be true
plan	trigger : guard ← body	*trigger* is the addition of a goal or a belief; *guard* gives conditions under which the plan can be executed; *body* is a stack of actions that should be performed
guard condition	B x	checking if belief x is perceivable
guard condition	G x	checking if goal x has been added
action	perf(x)	action x causing the execution of x

Fig. 2. Gwendolen [8] syntax.

while both Abstraction and Decision-Making Agents are programmed in the Gwendolen programming language. An interface between TORCS and MATLAB/Simulink has been developed that provides a means to control vehicles from MATLAB and Simulink.[4] In Section 3.1, we overview the Gwendolen programming language along with the agent code for the joining scenario.

The Decision-Making Agent makes high-level decisions based on its beliefs, goals, etc. These decisions are at the level of *agreement*, *coarse vehicle commands*, *communications*, etc., and are translated, by the Abstraction Agent, to provide input for the Physical Engine. As the Physical Engine runs, real data and activity is translated, again by the Abstraction Agent, into beliefs for the Decision-Making Agent. An important part of the design process involves ensuring that all relevant high-level commands and lower level controls are matched.

3.1. Gwendolen *programming language*

Gwendolen is a declarative logic-programming language incorporating explicit representations of goals, beliefs, and plans. Listing 1 shows some of the Gwendolen code from the decision-making agent for the joining procedure for a follower vehicle.[5]

Gwendolen uses many syntactic conventions from BDI agent languages. We describe those needed to understand the example (the syntax is also summarised in Fig. 2). : +!g indicates the addition of the goal g; +b indicates the addition of the belief b; while -b indicates the removal of the belief. ∗b indicates that execution of a plan is suspended until the belief, *b*, becomes true. Plans consist of three parts, with the pattern

```
trigger: guard <— body;
```

The 'trigger' is the addition of a goal or a belief (beliefs may be acquired thanks to the operation of perception and as a result of internal deliberation); the 'guard' states conditions about the agent's beliefs and goals which must be true before the plan can become active; and the 'body' is a stack of 'deeds' the agent performs in order to execute the plan. These deeds typically involve the addition and deletion of goals and beliefs as well as *actions* (e.g., perf(changing_lane(1))) which refer to code that is delegated to non-rational parts of the systems. Plans may contain uninstantiated variables which we represent with upper-case letters, as is common in logic programming languages. When a plan's trigger is matched to a new belief or goal this will generate a unifier that instantiates any variables in the trigger, and evaluation of the plan's guard may instantiate further variables. This unifier is then applied to the body of the plan and this body is added to an *intention* that already contained the trigger event.

A full operational semantics for Gwendolen is available in [7,8] but in brief a Gwendolen agent has a *reasoning cycle* that follows the characteristics of reasoning outlined by the BDI architecture [33]. The agent moves through this cycle polling an external environment for perceptions and messages; converting these into beliefs and creating intentions from any new beliefs; selecting an intention for consideration (by default intentions are stored as a queue and the first in the queue is selected); if the intention has no associated plan body then the agent looks for a plan that matches the trigger event, extracted from the environment, (and by default selects the first from the program that applies) and places the body of this plan on the intention; the agent then processes the first deed on the intention (e.g., adding or removing a belief, adding or removing a sub-goal, or performing an action) and then places the intention at the end of the intention queue before performing perception once more.

Plan guards are evaluated using Prolog-style reasoning. The statement B b is true if either b unifies with a ground literal in the agent's belief base, or there exists a *reasoning rule* of the form h :— body where h unifies with b and body is a conjunction of literals which in turn either unify with a literal in the agent's belief base or can be deduced via further reasoning rules. Similarly the statement G g is true if *g* unifies with a ground literal in the agent's goal base, or there exists a reasoning rule from which it can be deduced. Negation is indicated with ~ and its semantics are negation by failure as in Prolog. Listing 1 contains one reasoning rule (line 3) which allows the agent to deduce the truth of joining(X, Y) for some X and Y if the agent's name is X and it believes platoon—ok (a belief that is asserted by the final plan in the listing).

[4] TORCSLink – The interface between TORCS and Simulink http://dx.doi.org/10.5281/zenodo.13943.

[5] The complete agent code is available at https://github.com/VerifiableAutonomy/AgentPlatooning.

Listing 1 A follower vehicle's code.

```
 1
 2  Reasoning Rules
 3  joining(X, Y):— name(X), platoon—ok
 4
 5  Plans
 6  +! joining(X, Y) [achieve]: {B name(X) , ~B join_agreement(X, Y)}
 7      <— +!speed_contr(0) [perform], +!steering_contr(0) [perform],
 8         .send(leader, :tell, join_req(X, Y)), *join_agreement(X, Y);
 9
10  +! joining(X, Y) [achieve]: {B name(X), B join_agreement(X, Y),
11      ~B changed_lane, ~G set_spacing(Z) [achieve]}
12      <— +!speed_contr(0) [perform], +!steering_contr(0) [perform],
13         perf(changing_lane(1)), *changed_lane;
14
15  +! joining(X, Y) [achieve]: {B name(X), B join_agreement(X, Y),
16      B changed_lane, ~B speed_contr, ~ B steering_contr,
17      ~B joining_distance, ~G set_spacing(Z) [achieve]}
18      <— +!speed_contr(1) [perform], *joining_distance;
19
20  +! joining(X, Y) [achieve]: {B name(X), B join_agreement(X, Y),
21      B changed_lane, B speed_contr, ~B steering_contr,
22      B joining_distance, ~G set_spacing(Z) [achieve]}
23      <— +!steering_contr(1) [perform];
24
25  +! joining(X, Y) [achieve]: {B name(X), B join_agreement(X, Y),
26      B changed_lane, ~B speed_contr, ~B steering_contr,
27      B joining_distance, ~G set_spacing(Z) [achieve]}
28      <— +!speed_contr(1) [perform], +!steering_contr(1) [perform];
29
30  +! joining(X, Y) [achieve]: {B name(X), B join_agreement(X, Y),
31      B changed_lane, B speed_contr, B steering_contr,
32      B joining_distance, ~B platoon_m, ~G set_spacing(Z) [achieve]}
33      <— .send(leader,:tell,message(X,joined_succ),
34         *platoon_m, +platoon—ok;
```

Thus the first plan in Listing 1 (Lines 6–8) states that if the follower agent gains a goal to join the platoon (+! joining(X, Y)) and its name is X (B name(X)) and it does not yet have a joining agreement (~B join_agreement(X, Y)), then it sets its speed and steering controls into state 0 (manual control) and sends a request to the leader to join the platoon. It then waits for an agreement to be received.

Gwendolen uses two types of goals, *achieve* goals and *perform* goals (indicated by the keywords [achieve] and [perform] in plans). An achieve goal indicates a state of the world that the agent wishes to bring about and these goals persist until they are achieved. So, in the case of the first plan, after the join agreement is received another plan may be attempted for the trigger +! joining(X, Y) if the belief joining(X, Y) is not yet deducible from agent's belief base and reasoning rules. Perform goals, on the other hand, represent sub-procedures. They are matched to a plan which is then executed and the goal removed without any check that they have succeeded. +! speed_contr(0) and +! steering_contr(0) in the first plan are perform goals – the agent simply enacts the relevant actions required to switch the control systems.

The second plan in Listing 1 (Lines 10–13) can be executed if, and only if, the follower agent believes it has a join agreement but does not yet believe that the agent has successfully changed lane nor that the agent currently has a goal to set a particular spacing between cars. Each of the subsequent plans in Listing 1 is executed in a slightly different circumstance. When a lane change goes smoothly, each plan gets executed at most once. However if something goes wrong, for instance the join agreement is revoked halfway through the manoeuvre, then an earlier plan may be executed again (e.g., if the join agreement is revoked then the agent returns the speed and steering controls to manual and requests a new join agreement from the leader).

The protocol embodied by the plans for achieving joining(X, Y) is that a lane change manoeuvre starts with manual control of speed and steering. A request to join the platoon is sent. When an agreement is received a lane change manoeuvre is performed. Once the lane is successfully changed there are two options, if the car is the appropriate distance from the one in front the automated systems take control of both speed and steering. If the car is not at the appropriate distance then the automated systems take control of the speed until an appropriate distance is achieved and then take control of the steering. At this point (the car is in the correct lane, the correct distance from the car in front and the automated systems are in full control) the agent sends a message to the leader confirming a successful manoeuvre, awaits receipt of confirmation from the leader and then asserts an internal belief that the platoon is now OK.

The BDI model is a useful paradigm for goal-oriented scenarios such as automotive platooning as the system reasons over different goals and intelligently selects plans to achieve them, removes and adds new goals, and through the execution of plans can commit to its intentions for achieving goals. This makes the BDI model a suitable approach for making high-level decisions in systems with complex goals, employed in dynamically changing environment. However, a limitation of the BDI model is in generation of new plans on demand. This is not part of the core approach but can be tackled, for example,

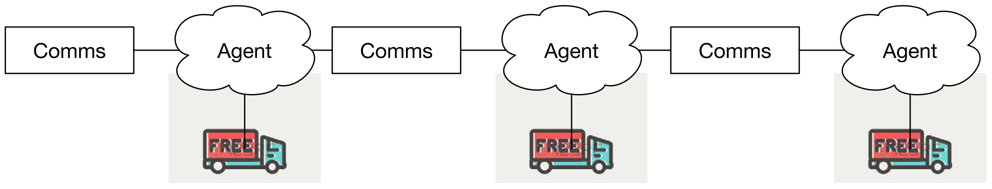


Fig. 3. Overall platoon system.

by integrating a module for the generation of plans via partial observable Markov decision process (POMDP) with the BDI model [28]. However, this is out of our paper's scope and we assume the agent's plan library is fixed.

4. Verification

This section presents the verification of the automotive platooning. We start by overviewing our verification methodology in Section 4.1 where the verification task is divided to two main parts: agent behaviour and real-time requirement of the system. We apply our methodology for verification of automotive platoon in the rest of sub-sections. In Section 4.2, the agent behaviour is verified using AJPF. To verify the real-time requirement of the system, we use the Uppaal model checker where a system is represented as timed automata. To generate timed automata for the system, first in Section 4.3 we introduce a translation algorithm that generates a timed automaton from an agent model by abstracting the internal states of the agent. Then in Section 4.4, we describe our Uppaal model of the system followed with its verification in Section 4.5. In the last part of the section, we discuss the validation of verification.

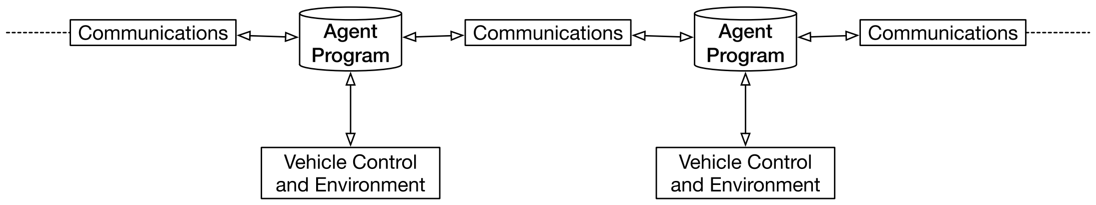
4.1. Verification methodology

We can visualise the overall platoon system as shown in Fig. 3.

The agent is a Gwendolen program, the *Comms* component is a simple transfer protocol, and the vehicle represents the particular vehicular system that we interact with. This is typically an automotive control system together with environmental interactions, and we have validated this both in simulation (using the TORCS automotive simulation) and in physical vehicles (using Jaguar outdoor rover vehicles).

Limits to modelling/verification. We are not going to formally verify the vehicular control systems, and leave this to standard mathematical (usually analytic) techniques from the Control Systems field [27]. These control components, for example involving following a prescribed path, avoiding local obstacles, keeping distance from objects, etc., are well-established and standard. Instead, we will verify the autonomous *decisions* the vehicles make, captured within each vehicle's 'agent' [16]. Each agent represents the autonomous decision-maker within each vehicle and corresponds, in part, to the human driver's decisions. These decisions involve deciding *where to go*, *when to turn*, *when to stop*, *what to do in unexpected situations*, etc. In the case of autonomous vehicle convoys/platoons, the agent's (and, hence, the vehicle's) decisions concern *when to join the convoy*, *when to leave*, *what to do in an emergency*, etc.

So, we begin by abstracting from all the vehicle control systems and environmental interactions, representing these by one (potentially complex, depending on the vehicle/environment interactions) automaton. We also use an automaton to describe the simple transfer protocol that the vehicles use for their communication. In both these cases we will use *Timed Automata* [1]. Simplified, in the sense that the *abstraction aspects*, *continuous control*, and *environmental considerations* are all bundled in to "Vehicle Control and Environment", our architecture is:



which, abstracts to an overarching formal model:

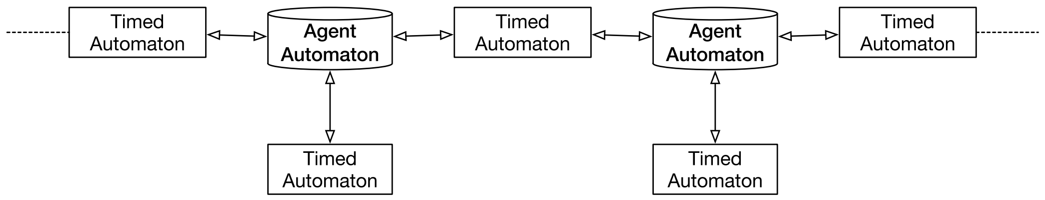
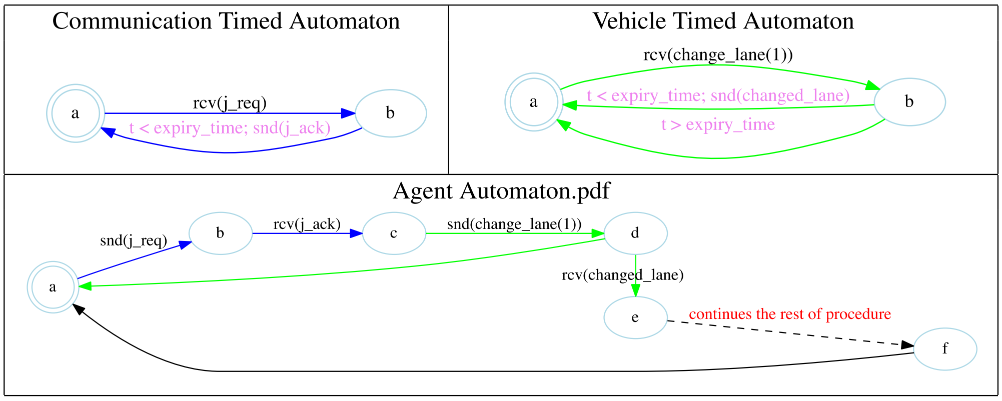


Table 1
An example of timed and untimed automata of the platooning.



Simplified timed automata of our architecture are given in Table 1.[6] The *communication* automaton represents the required communications for an agent to receive a join agreement namely, receiving a join request from an agent and sending back a join agreement. The *vehicle* automaton represents the behaviour of a vehicle when it receives a "changing lane" command from the agent. The *agent* automaton represents a simplified behaviour of an agent for joining to the platoon. Transitions between locations are synchronised through *rcv* and *snd* channels. For instance, when the agent automaton send a join request ($snd(j\_req)$), it synchronises with $rcv(j\_req)$ of the communication automaton.

In describing agent behaviour, the agent automaton incorporates the dimensions of *belief* and *intention*, allowing representation of rational agent behaviour. Thus, the formal structures that allow us to fully represent the whole system above are quite complex, combining timed relations as well as relations for each of the belief and intention dimensions [31,1]. We will not describe this formal model in detail but just note that it is a *fusion* [15,17,25] of timed and BDI structures, $\langle L, A, C, \mathcal{E}, inv, R_B, R_I, l\rangle$. Here: L is a finite set of locations; A is a finite set of actions; C is a finite set of clocks, for the timed aspect; $\mathcal{E} \subseteq L \times \Psi(C) \times A \times 2^C \times L$ is a set of (timed) edges between locations, where $\Psi(C)$ is a set of simple clock constraints such as "$c < 4$"; $inv : L \to \Psi(C)$ is a function associating each location with some clock constraint in $\Psi(C)$; $R_B : Ag \to (L \times L)$ provides the belief relation between locations for agents in Ag; $R_I : Ag \to (L \times L)$, intention relation for agents between locations; and $l : L \to 2^{AP}$ is a labelling function essentially capturing those propositions true at each location (AP is a set of atomic propositions).

The logic is then interpreted over such structures combining [17] the syntax of timed temporal logic, for example $\Diamond^{\leq 5}$*finish* (i.e., "eventually within 5 seconds *finish* holds"), and the syntax of modal logics of belief, desire and intention, for example B_x*started* (i.e. "agent x believes that *started* is true").

In principle, though very complex, we could provide all our platoon requirements in such a logic, build structures of the above form for our platoon implementation, and then develop a model-checking approach for this combination [24]. However, there are several reasons we choose to abstract and separate the timed/agent strands, as follows.

(I) For *certification* it is important that we verify the *actual* agent program used in each vehicle, not a derived model of this. Consequently, we utilise a *program model checking* [38] approach to assess the correctness of each agent program. Program model checking analyses the actual code in detail rather than generating a further finite state model to represent it. For this formal verification of the agent's autonomous decisions, we use AJPF [12], an extension of the Java PathFinder (JPF) program model checker [23] for Gwendolen that allows verification of belief/intention properties. This is the only program model-checker for rational agents.

(II) We do not have the detailed implementations of all the communication protocols, the vehicular controls, and environmental interactions, and so must use an abstract, formal model to describe these, rather than actual code. This allows us to narrow the verification to only the agent program but with the cost of losing the generality of results. Our abstraction dictates what the above components should do without looking at how they perform those actions.

(III) JPF is an explicit-state program-checker and is relatively slow; AJPF builds a BDI programming layer on top of JPF and is at least an order of magnitude slower than JPF. Consequently, AJPF cannot realistically be used for verification of the whole system. In addition, as AJPF does not yet have real-time capabilities, then verifying timing aspects within AJPF is difficult.

While these appear problematic, there are several useful simplifications in our context:

[6] The full model of timued automata is available at https://github.com/VerifiableAutonomy/AgentPlatooning.

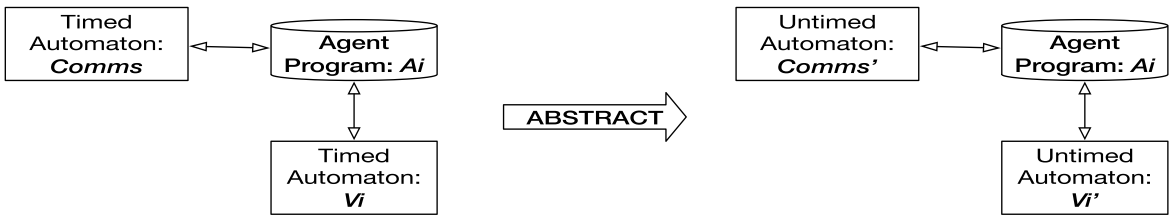
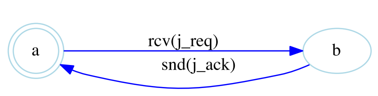
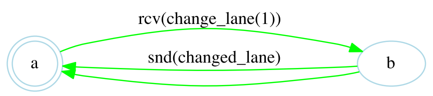


Fig. 4. Timed automaton to untimed automaton.

Table 2
An example of untimed automata of *Comm* and V_i.

Communication Untimed Automaton	Vehicle Untimed Automaton
	

- When verifying autonomous behaviour, the formal verification we carry out concerns the interaction of beliefs, intentions, etc., *within* each agent. These do not extend between agents and so, checking of beliefs, intentions, etc, can be localised within each agent.
- In the requirements to be checked, the timed and BDI formulae are quite separate, i.e. $\Diamond^{\leq 5}\mathit{finish} \wedge B_x \mathit{started}$ but never $\Diamond^{\leq 5} B_x \mathit{started}$ or $B_x \Diamond^{\leq 5}\mathit{finish}$. As the overall logic is a fusion, and since there are no explicit timing constraints *within* an agent program (agents have fast internal computation), this allows us to deal with the timing and agent dimensions separately.

So, given an overall system, S, over which we wish to check φ, then we reduce $S \models \varphi$ to two problems:

1. for each individual agent, A_i, within S, verify the agent properties from φ, i.e. φ_a, on the agent within an untimed environment (an over-approximation) which addresses the concerns given in (I) and (III); and
2. verify the timing properties from φ i.e. φ_t, on the whole system where the agent program is replaced by a timed automaton describing solely its *input–output* behaviour (abstracting from internal BDI reasoning) which addresses the problems given in (II).

We explain both of these in more detail, before giving the relevant theorems.

Timed automaton $\longrightarrow$ *untimed automaton.* This is essentially achieved by over-approximation. By removing timing constraints from automata transitions, we can still move between the same states but there are no timing restrictions on this movement. If we do this systematically, as in Fig. 4, then the new system represents an over-approximation of the original as more behaviours are allowable.

Note that going from a timed automaton to an untimed one, for example replacing behaviours such as $\Diamond^{\leq 3}\mathit{receive}$ by $\Diamond\mathit{receive}$ provides this over-approximation. Hence, for example, $\mathit{Comms}' \models \varphi$ implies $\mathit{Comms} \models \varphi$, but not necessarily the converse. This then allows us to verify[7] $V_i' \| \mathit{Comms}' \| A_i \models \varphi_a$ using AJPF as detailed timing is removed in the vehicle and communications models.

Taking the example in Table 1, the untimed automata in Table 2 are the result of timing removal for the *communication* and *vehicle* timed automata. Note that the Comms' and V_i' communicates with A_i through message passing.

Agent model $\longrightarrow$ *untimed automaton.* Formal verification, via model checking, of an agent program [11] carries out by exploring all possible states of an agent model. While the exploration is performing, a model of agent behaviour can be gardually built. This model is a BDI automaton incorporating all the temporal, belief, and intention aspects of the agent. Now we will abstract from the agent aspects to build a simple automaton where all the belief/intention updates are ignored. This then just corresponds to the basic I/O behaviour of the agent. See Fig. 5.

Formal verification using Uppaal can then carried out on the whole system with all agents abstracted by substituting with untimed automata. In this abstraction, Comms and V_i communicate with A_i' through synchronisation channels.

We now provide proof sketches for properties of the above abstractions. For simplicity, we assume that S consists of just two agents/vehicles; this result can then easily be generalised to greater numbers of agents/vehicles. Below, recall that:

- V_1 and V_2 are timed automata representing the vehicle control, while V_1' and V_2' are untimed abstractions of these;

[7] Note that the parallel composition operator $\|$ is deliberately generic to simplify description.

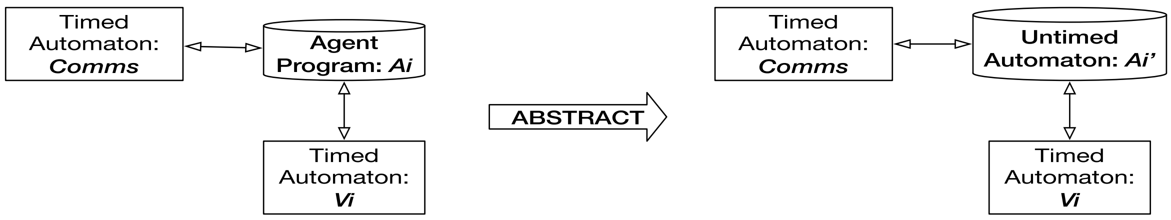


Fig. 5. Agent model to untimed automaton.

- A_1 and A_2 are BDI automata representing the agents making decisions, while A'_1 and A'_2 are abstractions of these with BDI elements removed; and
- *Comms12* is a timed automaton representing the inter-vehicle communications, while *Comms12'* is an untimed abstraction of this (a standard Büchi automaton).

Theorem 1. *Let* $S == V_1 \parallel A_1 \parallel Comms12 \parallel A_2 \parallel V_2$. *If*

a) $V'_1 \parallel A_1 \parallel Comms12' \models \varphi_a$ *and*
b) $V'_2 \parallel A_2 \parallel Comms12' \models \varphi_a$ *and*
c) $V_1 \parallel A'_1 \parallel Comms12 \parallel A'_2 \parallel V_2 \models \varphi_t$.

then $S \models \varphi_a \wedge \varphi_t$.

Proof Sketch. Since V'_1 and *Comms12'* are over-approximations, the behaviours of V'_1 include all behaviours of V_1 and the behaviours of *Comms12'* include all behaviours of *Comms12*, and so

$$V'_1 \| A_1 \| Comms12' \models \varphi_a \text{ implies } V_1 \| A_1 \| Comms12 \models \varphi_a.$$

Similarly, (b) gives us $V_2 \| A_2 \| Comms12 \models \varphi_a$. As the agent properties in φ_a are local, we can compose these to give $V_1 \| A_1 \| Comms12 \| A_2 \| V_2 \models \varphi_a$ and so $S \models \varphi_a$.

By (c) we know that $V_1 \| A'_1 \| Comms12 \| A'_2 \| V_2 \models \varphi_t$ yet, contain no timing constraints to begin with, we know that A'_1 and A'_2 give us exactly the same *timed* behaviours. Thus A_1 and A'_1 are equivalent wrt φ_t, as are A_2 and A'_2. Consequently, $V_1 \| A_1 \| Comms12 \| A_2 \| V_2 \models \varphi_t$ and so $S \models \varphi_t$. These two together give us $S \models \varphi_a \wedge \varphi_t$. □

Theorem 2. *If* $V_1 \parallel A_1 \parallel Comms12 \parallel A_2 \parallel V_2 \models \varphi_t$ *then* $V_1 \parallel A'_1 \parallel Comms12 \parallel A'_2 \parallel V_2 \models \varphi_t$.

Proof Sketch. Since the timing behaviour of each A_i is identical to each A'_i then

$$V_1 \| A'_1 \| Comms12 \| A'_2 \| V_2$$

and

$$V_1 \| A_1 \| Comms12 \| A_2 \| V_2$$

are equivalent. □

As we mentioned earlier, V'_1, V'_2, and *Comms12'* are over-approximation, and so

$$V_1 \| A_1 \| Comms12 \| A_2 \| V_2 \models \varphi_a \text{ does not necessarily imply}$$
$$V'_1 \| A_1 \| Comms12' \| A_2 \| V'_2 \models \varphi_a.$$

In summary, we abstract from timing behaviours in vehicle and communication models so that we can carry out (untimed) verification of these in combination with the agent, and (separately) abstract form detailed BDI behaviours in the agent so we can verify timed behaviours of the combination of original communications model, original vehicle model, and abstracted agent. This is feasible since, in our application, BDI behaviours only occur *within* agents and the agent itself does *not* contribute to timing delays.

4.2. Individual agent verification using AJPF

To verify individual agent properties, we use the AJPF model checker on our agent, written in the Gwendolen language, as above. For instance, we verify that:

If a vehicle never believes it has received confirmation from the leader, then it never initiates joining to the platoon.

This safety property corresponds to the first requirement of joining a platoon, as given in Section 2, and can be defined as:

$$\Box\,(\mathrm{G}_{f3}\ \mathtt{platoon\_m}(f3, f1) \rightarrow \neg\mathrm{D}_{f3}\ \mathtt{perf(changing\_lane(1))}\ W\ \mathrm{B}_{f3}\ \mathtt{join\_agr}(f3, f1)) \tag{1}$$

Here f3 refers to a non-member vehicle which tries to join the platoon, in front of member vehicle f1. $G_x\ y$ stands for a goal y that agent x tries to achieve, $B_x\ z$ stands for a belief z of agent x, and $D_x\ k$ stands for an action k that agent x performs/does. The standard LTL operators, such as $\Box$ meaning "always in the future" and W meaning "unless", are used. An instance of the above where the agent *never* receives a join agreement, is:

$$\Box\,(\mathrm{G}_{f3}\ \mathtt{platoon\_m}(f3, f1)\ \&\ \neg\mathrm{B}_{f3}\ \mathtt{join\_agr}(f3, f1)) \rightarrow \Box\,\neg\mathrm{D}_{f3}\ \mathtt{perf(changing\_lane(1))} \tag{2}$$

To be able to check such a property, incoming perceptions and communications should be provided. One way to provide such inputs is to generate them randomly. However, the whole combination of incoming perceptions and communications is significant and consequently the property checking is infeasible. Another way to provide incoming perceptions and communications is to represent input combinations that are viable from the practical vehicle behaviour. This often makes the property checking feasible. Thus, we supply two automata: $Comm'$, representing communication to and from the other agents; and V_i', representing vehicle responses to agent actions. The latter solution for providing incoming perceptions and communications, on one hand, removes all possibilities for providing irrelevant inputs by mimicking a real model of the communication and vehicle controller and decreases the verification time. On the other hand, it does not ensure the correctness of the agent code against completely unanticipated input. Under this configuration, we were able to carry out the agent verification in around 20 minutes. (Recall that AJPF is *very* resource hungry.) Note that we can, with additional resources, weaken expectations on the environment and so verify the agent in increasingly unlikely scenarios.

We have verified a range of safety and liveness properties and we provide some joining and leaving examples below. Note that the following properties can also be similarly expressed in terms of the weak until operator, W; however, we denote a particular instance of these properties for the sake of brevity.

If a vehicle ever sends a 'join' request to the leader and eventually receives the join agreement and it is not already in the correct lane, it initiates 'joining' the platoon by performing "changing lane".

$$\Box\,(\mathrm{G}_{f3}\ \mathtt{platoon\_m}(f3, f1)\ \&\ \neg\mathrm{B}_{f3}\ \mathtt{changed\_lane}\ \&\ \Box\,(\mathtt{ItD}_{f3}\mathtt{send(leader, tell, message}(f3, 1, f1)) \rightarrow \Diamond\,\mathrm{B}_{f3}\ \mathtt{join\_agr}(f3, f1))) \rightarrow \Diamond\,\mathrm{D}_{f3}\ \mathtt{perf(changing\_lane(1))} \tag{3}$$

Property 3 is a liveness property ensuring that eventually the joining procedure initiates the changing lane control system once its condition is fulfilled. Similarly, we can verify other properties to show progress such as "eventually the speed and steering controllers are switched to automatic if pre-conditions hold". Some other verified properties ensuring safe operation of the platoon are as follows.

If a vehicle never believes it has changed its lane, then it never switches to the automatic speed controller.

$$\Box\,(\mathrm{G}_{f3}\ \mathtt{platoon\_m}(f3, f1)\ \&\ \neg\mathrm{B}_{f3}\ \mathtt{changed\_lane}) \rightarrow \Box\,\neg\mathrm{D}_{f3}\ \mathtt{perf(speed\_controller(1))} \tag{4}$$

If a vehicle never believes it has received a confirmation from the leader, then it never switches to the automatic speed controller.

$$\Box\,(\mathrm{G}_{f3}\ \mathtt{platoon\_m}(f3, f1)\ \&\ \neg\mathrm{B}_{f3}\ \mathtt{join\_agr}(f3, f1)) \rightarrow \Box\,\neg\mathrm{D}_{f3}\ \mathtt{perf(speed\_controller(1))} \tag{5}$$

If a vehicle never believes it is sufficiently close to the preceding vehicle, it never switches to the automatic steering controller.

$$\Box\,(\mathrm{G}_{f3}\ \mathtt{platoon\_m}(f3, f1)\ \&\ \neg\mathrm{B}_{f3}\ \mathtt{joining\_distance}) \rightarrow \Box\,\neg\mathrm{D}_{f3}\ \mathtt{perf(steering\_controller(1))} \tag{6}$$

If a vehicle never believes it has received a confirmation from the leader to leave the platoon, i.e., increasing spacing has been achieved, then it never disables its autonomous control.

Note that the leader sends back the 'leave' agreement to *follower3* if, and only if, it received an acknowledgement from *follower3* showing that spacing has been increased.

$$\begin{aligned}&\Box\,(\mathrm{G}_{f3}\;\texttt{leave\_platoon}\;\&\;\neg\mathrm{B}_{f3}\;\texttt{leave\_agr}\,(\texttt{f3}))\\ &\rightarrow \Box\,\neg\mathrm{D}_{f3}\;\texttt{perf(speed\_controller(0))}\end{aligned} \tag{7}$$

It is important to recall that perceptions and communications coming in to the agent are represented as internal beliefs. Hence the proliferation of belief operators. The AJPF program model checker explores all possible combinations of shared beliefs and messages and so, even with the relatively low number of perceptions above, the combinatorial explosion associated with exploring all possibilities is very significant. Therefore, verifying the whole multi-agent platooning system using AJPF is infeasible.

As explained in Section 4.1, to verify the global properties of multi-agent platooning, we use a complementary approach. We generate a model of the whole system as timed-automata and use the Uppaal model checker to establish the (timed) correctness of multi-agent platooning. In the following, we review the relevant timed-automata and highlight some of the global safety properties of vehicle platooning that have been verified using Uppaal.

4.3. From agent model to untimed automata

In this section we define our algorithm for extracting an untimed automaton from an agent model. First, we generate a state model of an agent model through the execution of the AJPF model checker. In every state we have a set of beliefs, intentions, actions, and messages which can represent either the internal state of the agent, or the input/output information from/to the environment. The environment is an abstract representation of a vehicle and other agents, in this context. For automaton generation only the input/output information is required. Therefore, we remove superfluous information from the state model and merge equivalent states, as described in Algorithm 1. Our approach is similar to the automata minimisation algorithms [21,22], where equivalent states are identified for a deterministic finite automata A. Equivalent states are merged to build the states of the minimal automaton A'.

The StripInternalBeliefs function updates the set of states (S) by firstly removing all internal beliefs and intentions, and then converting beliefs and actions representing input/output information to propositions. Typically, this replaces belief formulae such as $B\ xyz$ by propositions such as *bel_xyz*. The RemoveEmpty function checks the propositions of states. For every empty state, the set of edges (E) is updated by redirecting the incoming and outgoing edges of the empty state. The set of states (S) is updated by removing the isolated empty state. The MatchAndMergeDuplicates function finds all states having equivalent outgoing edges and propositions. The set of edges (E) is updated by redirecting the incoming edges of equivalent states. The isolated equivalent state is then removed from the set of states.

The agent model consists of 28 beliefs, 14 actions/messages, and 12 goals. In total, 22 beliefs, actions, and messages can be converted to propositions that represent input/output information. The generated state model of the agent code consists of 1396 states and 2361 transitions. The failure of the communication component, represented by the *Comms*′ automaton, has not been considered in the generation of the state model. Reliable communication plays a key role in safety of platooning. If the communication is unreliable, the platoon dissolves and consequently any joining and leaving operation will be stopped. In other words, we can only guarantee the correctness of joining and leaving protocols under reliable communication condition. In contrast, some failures of the vehicle (represented as the V_i' automaton), namely the failure to change lanes, have been included in the model. Changing lane is a procedure that a driver performs and can go wrong due to human error. Therefore, in order to have a realistic design, we consider human error. As the platooning scenarios are sequential and involve disjoint sets of events, a generated automaton from the agent model should consist of at least 22 states.

The output of Algorithm 1 shows that 650 states, out of 1396 generated states of the agent code, are empty states which are removed by the RemoveEmpty function. The MatchAndMergeDuplicates function reduces the number of states and transitions to 174 and 519, respectively. To optimise the output of Algorithm 1, two more reduction functions are introduced that are shown in Fig. 6. The first function finds states whose outgoing edges are to states with equivalent propositions, and redirects all the incoming edges and removes isolated states. Two states 129 and 145 in Fig. 6 (2–3) are merged with 127 using this reduction. The second function finds cases as shown in Fig. 6 (4) where a state with equivalent propositions, i.e., state 126, redirects its incoming state 125 to state 127 which is directly connected to 125. As we do not take into account timing aspects here, moving between identical states, then these can be merged. This simplification reduces the number of states and transitions to 89 and 560, respectively. The reduction functions are repeatedly executed until the numbering of states converges to a size of 34.

The Uppaal agent automaton shown in Fig. 7 is composed of 29 locations which are extracted from the output of Algorithm 1. Note that the Uppaal agent automaton has been generated manually since the generated state transition by AJPF had to be adapted to Uppaal locations and transitions. For instance, AJPF generates states at the end of each *reasoning cycle*, i.e., a generated state can consist of new beliefs and a performing action. This means that a state generated form the

Algorithm 1 Agent model to untimed automaton algorithm.

```
1: translateAgentToAutomaton($S, E$)
Input: a state model $A$, representing as a graph $G = (S, E)$
Output: a minimised state model $A'$, representing as $(S''', E''')$
2: $(S', E) = $ StripInternalBeliefs$(S, E)$
3: $(S'', E'') = $ RemoveEmpty$(S', E))$
4: $(S''', E''') = $ MatchAndMergeDuplicates$(S'', E'')$
5: function StripInternalBeliefs$(S, E)$
6:     $S' = \emptyset$
7:     for each $s \in S$ do
8:         $s' = s$ with all internal beliefs and intentions removed
9:         $s'' = s'$ with all remaining beliefs and messages converted to propositions
10:        $S' = S' \cup \{s''\}$
11:    end for
12:    return $(S', E)$
13: end function
14: function RemoveEmpty$(S, E)$
15:    $S' = S,\ E' = E$
16:    for each $s \in S'$ do
17:        if prop$(s) = \emptyset$ then
18:            $E' = (E' \cup \{(x, y) \mid x \in \mathbf{in}(s) \wedge y \in \mathbf{out}(s)\})$
                   $\setminus(\{(x, s) \mid x \in \mathbf{in}(s)\} \cup \{(s, y) \mid y \in \mathbf{out}(s)\})$
19:            $S' = S' \setminus \{s\}$
20:        end if
21:    end for
22:    return $(S', E')$
23: end function
24: function MatchAndMergeDuplicates$(S, E)$
25:    $S' = S,\ E' = E$
26:    for each $s_i \in S'$ do
27:        for each $s_j \in S' \setminus \{s_i\}$ do
28:            if out$(s_i) = $ out$(s_j)$ and prop$(s_i) = $ prop$(s_j)$ then
29:                $E' = (E' \cup \{(s_{in}, s_i) \mid s_{in} \in \mathbf{in}(s_j)\})$
                       $\setminus(\{(s_j, s_{out}) \mid s_{out} \in \mathbf{out}(s_j)\} \cup \{(s_{in}, s_j) \mid s_{in} \in \mathbf{in}(s_j)\})$
30:                $S' = S' \setminus \{s_j\}$
31:            end if
32:        end for
33:    end for
34:    return $(S', E')$
35: end function
```

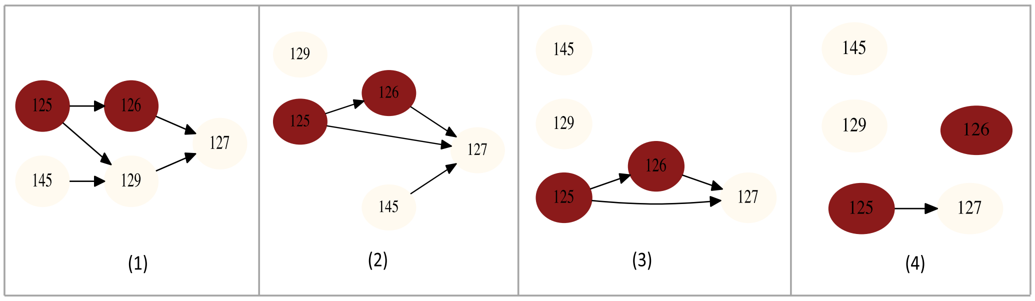


Fig. 6. A reduction example.

agent code can be transformed to two consecutive locations in Uppaal with two synchronisation channels which represent a new belief and a performing action. Furthermore, the output of Algorithm 1 is an untimed automaton that needs to be supplemented with timing requirements.

4.4. Timed automata model of automotive platoons

We model vehicle platooning in Uppaal as a parallel composition of identical processes describing the behaviour of each individual vehicle in the platoon along with an extra process describing the behaviour of the platoon leader (the *leader* automaton). Each of these vehicle processes is a parallel composition of two timed automata, the *vehicle control and environment* and *agent*. The Uppaal *agent* automaton,[8] in turn, comprises both *Comms* and A_i' components, as described in

[8] The full Uppaal model is available at https://github.com/VerifiableAutonomy/AgentPlatooning.

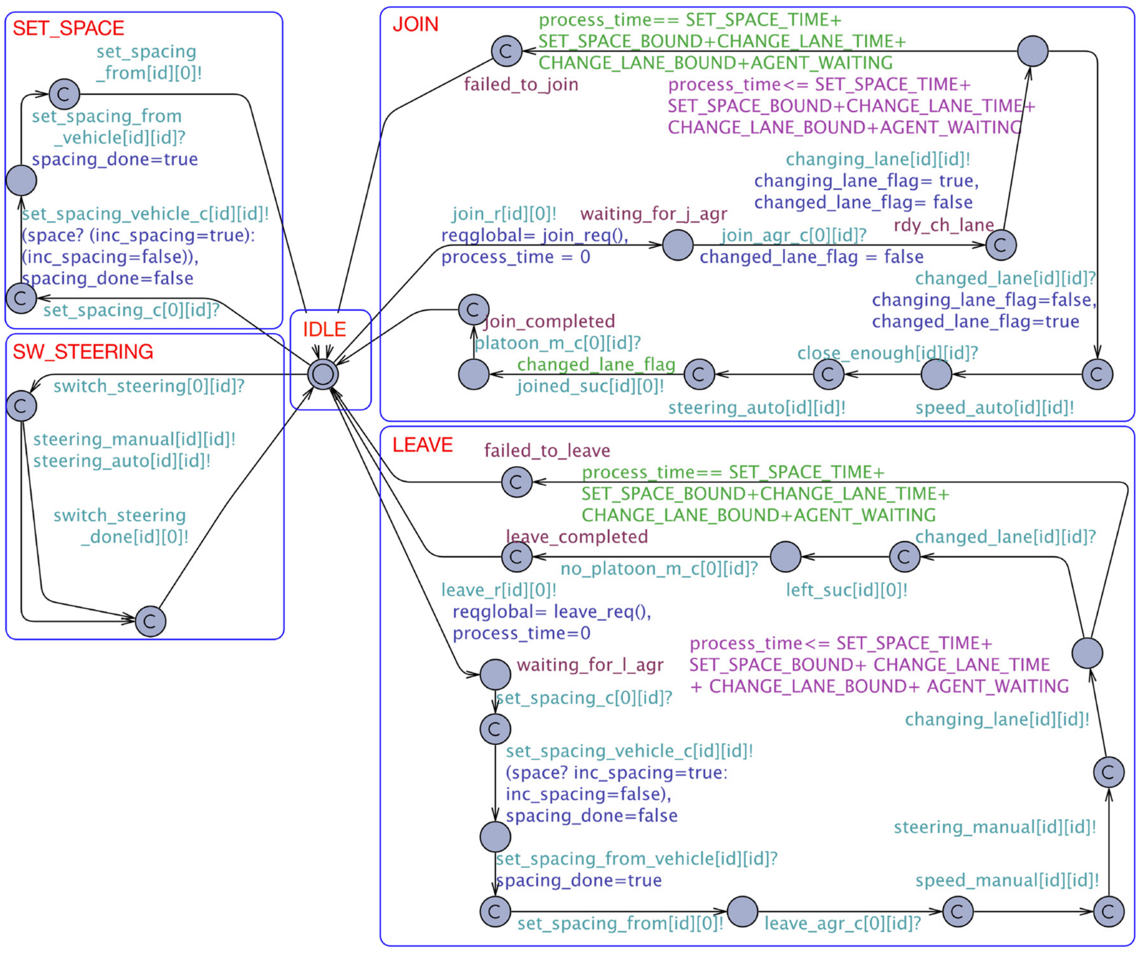


Fig. 7. Example Uppaal *agent* automaton.

Section 4.1. This means that the timing requirements of *Comms* component are added to untimed transitions of A'_i to model the Uppaal agent automaton.

The vehicle automaton supplies incoming perceptions for the Uppaal agent automaton. It describes the sensor models and action execution. The vehicle automaton receives, and responds to, the action commands of the corresponding agent through six pairs of binary channels modelling *change-lane*, *set-space*, *speed-manual/auto* and *steering-manual/auto* commands and responses. To model timing behaviour, we define a clock for the vehicle automaton which models the time assessments for “changing lane”, “setting space” and “speed auto” actions. Engineering analysis of the stability and robustness of the low level, continuous controllers has been used to ensure that these systems function correctly, within certain bounds [36, 27]. These bounds correspond to quantities such as acceleration and lateral velocity which subsequently impose constraints on the timing of decisions. Based on these analysis, actions *speed-manual* and *steering-manual/auto* happen immediately and actions *change-lane*, *set-space* and *speed-auto* take 20 ± 5, 10 ± 5 and 10 ± 5 seconds, respectively. For instance, the *speed-auto* action completes in 5 seconds in the best case and in 15 seconds in the worst case. It means when the Uppaal *agent* automaton sends a *speed-auto* command to the vehicle, it should wait between 5 to 15 seconds for *close-enough* response. Note that timing values are parameters of the timed automata and can be adjusted based on the type of vehicles. In other words, tuning the timing values in the timed automata model and system properties do not impact on the verification result of the automotive platoon.

The Uppaal *agent* automaton models an *abstracted* version of the Gwendolen agent by excluding all internal computations of the agent and added time constraints of *Comms*. The overall structure of an Uppaal agent consists of 5 regions, shown in Fig. 7. These regions are extracted from the output of Algorithm 1.

If the automaton is in the *IDLE* region, which consists of only one location, then the agent does not perform any action at that moment. The regions *JOIN*, *LEAVE*, *SET-SPACE* and *SW-STEERING* represent the sequence of necessary communications with other agents (and the vehicle) in order to achieve the agent’s goals. Each agent automaton contains two binary channels *join-r[i][0]* and *leave-r[i][0]* to model the unicast sending of ‘join’ and ‘leave’ requests to the leader and two binary channels *joined-suc[i][0]* and *left-suc[i][0]* to model the unicast sending of ‘join’ and ‘leave’ *acknowledgements* to the

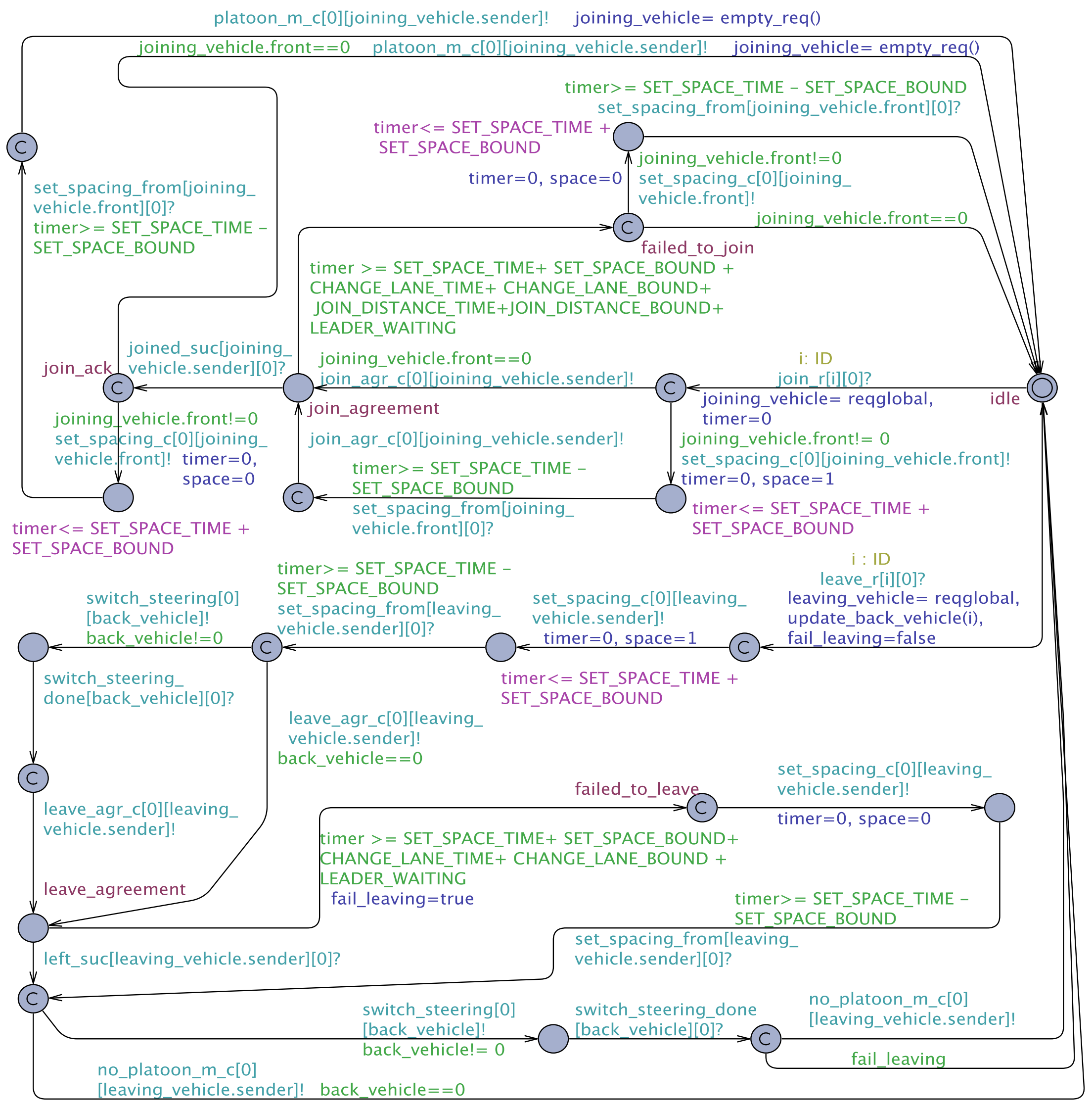

platoon_m_c[0][joining_vehicle.sender]!
joining_vehicle= empty_req()
joining_vehicle.front==0
platoon_m_c[0][joining_vehicle.sender]!
joining_vehicle= empty_req()
timer>= SET_SPACE_TIME – SET_SPACE_BOUND
set_spacing_from[joining_vehicle.front][0]?
timer<= SET_SPACE_TIME + SET_SPACE_BOUND
joining_vehicle.front!=0
set_spacing_c[0][joining_ vehicle.front]!
timer=0, space=0
set_spacing_from[joining_ vehicle.front][0]?
timer>= SET_SPACE_TIME – SET_SPACE_BOUND
joining_vehicle.front==0
failed_to_join
timer >= SET_SPACE_TIME+ SET_SPACE_BOUND + CHANGE_LANE_TIME+ CHANGE_LANE_BOUND+ JOIN_DISTANCE_TIME+JOIN_DISTANCE_BOUND+ LEADER_WAITING
joined_suc[joining_ vehicle.sender][0]?
join_ack
joining_vehicle.front==0
join_agr_c[0][joining_vehicle.sender]!
i: ID
join_r[i][0]?
joining_vehicle= reqglobal, timer=0
idle
join_agreement
joining_vehicle.front!=0
set_spacing_c[0][joining_ vehicle.front]!
timer=0, space=0
join_agr_c[0][joining_vehicle.sender]!
joining_vehicle.front!= 0
set_spacing_c[0][joining_vehicle.front]!
timer=0, space=1
timer>= SET_SPACE_TIME – SET_SPACE_BOUND
set_spacing_from[joining_ vehicle.front][0]?
timer<= SET_SPACE_TIME + SET_SPACE_BOUND
timer<= SET_SPACE_TIME + SET_SPACE_BOUND
timer>= SET_SPACE_TIME – SET_SPACE_BOUND
set_spacing_from[leaving_ vehicle.sender][0]?
switch_steering[0] [back_vehicle]!
back_vehicle!=0
set_spacing_c[0][leaving_ vehicle.sender]!
timer=0, space=1
i : ID
leave_r[i][0]?
leaving_vehicle= reqglobal, update_back_vehicle(i), fail_leaving=false
switch_steering_ done[back_vehicle][0]?
timer<= SET_SPACE_TIME + SET_SPACE_BOUND
leave_agr_c[0][leaving_ vehicle.sender]!
back_vehicle==0
failed_to_leave
set_spacing_c[0][leaving_ vehicle.sender]!
timer=0, space=0
leave_agr_c[0][leaving_ vehicle.sender]!
timer >= SET_SPACE_TIME+ SET_SPACE_BOUND+ CHANGE_LANE_TIME+ CHANGE_LANE_BOUND + LEADER_WAITING
fail_leaving=true
leave_agreement
timer>= SET_SPACE_TIME – SET_SPACE_BOUND
set_spacing_from[leaving_ vehicle.sender][0]?
left_suc[leaving_vehicle.sender][0]?
switch_steering[0] [back_vehicle]!
back_vehicle!= 0
switch_steering_done [back_vehicle][0]?
no_platoon_m_c[0] [leaving_vehicle.sender]!
fail_leaving
no_platoon_m_c[0] [leaving_vehicle.sender]!
back_vehicle==0

Fig. 8. The *leader* automaton.

leader. These channels are used to model the message passing between the following agents and the leader, modelled in the decision-making agent (Section 3). Furthermore, each agent automaton also contains channels to send commands to its vehicle and receive acknowledgements from its vehicle. The agent automaton has a clock *process-time* that is used to model the time consumption for achieving goals.

Next, we define a leader automaton to model the external behaviour of the leader agent (Fig. 8), where the coordination between agents is handled through unicast synchronisation channels. Upon receipt of a joining request, i.e., *join-r[i][0]!*, it sends a "set spacing" command to the preceding agent where the requested agent wants to be placed. The leader sends a joining agreement, i.e., *join-agr-c[0][i]?*, to the requested agent, if it has successfully set spacing between the two vehicles where the requested vehicle will be placed. Follower *i* synchronises with the leader via *join-agr-c* channel. Then the leader waits for an acknowledgement from the requested agent. It waits for at most the upper bound time for setting space, changing lane and getting close enough to the front vehicle. Upon receipt of the acknowledgement, the leader sends a confirmation to the agent and a "set spacing" command to the preceding agent to decrease its space with the front vehicle to complete the joining procedure. If it does not receive the acknowledgement in time, it sends a "set spacing" command to the preceding agent to decrease its space and waits for a spacing acknowledgement then goes back to the *idle* location, ready for the next request. The leader communicates with the agents through synchronisation channels. It passes messages to the follower through channels dedicated to the agreements, setting space and switching steering controller. For simplicity, we assume the leader handles only one request at any time, to avoid the unsafe situation of simultaneous joining and leaving operations that are described in Section 2.

4.5. Multi-agent platooning verification using Uppaal

Now we have timed automata representations of the platoon, we can carry out verification of their properties using Uppaal. For simplicity, we analyse the global and timing properties of a multi-agent platoon composed simply of a leader and three vehicles (with three corresponding) agents. However, the verification can perform for arbitrary length of platoon by simply instantiating more vehicles and agents. We assume vehicles can always set spacing and joining distance in time, i.e., 10 ± 5, but can fail to change lane in time, i.e., less than $20 + 5$. In the following, we first give examples of global properties involving the coordination between the leader and the followers. Second, we evaluate timing requirements: the safe lower and upper bounds for joining and leaving activities. We observed that the verification of these properties took less than 3 seconds using Uppaal. The reason is that the behaviour of the system is sequential and the interleaving is restricted in the *leader* automaton. Unreliable communication is not acceptable for automotive platoons and if such errors are detected the platoon is dissolved. Therefore, we did not consider communication errors in our automata; however, the changing lane failure can occur. The *leader* automaton detects the failure when it does not receive any acknowledgement in due time and readjusts the platoon and going back to *idle* state, i.e., ready for the next request.

If an agent ever receives a joining agreement from the leader, then the preceding agent has increased its space to its front agent. This property is formulated for agent *a3* as follows (**A** represents "on all paths"):

$$\begin{aligned}&\mathbf{A}\Box\ ((\texttt{a3.rdy\_ch\_lane}\ \mathbf{\&\&}\ \texttt{l.joining\_vehicle.front} == 2)\\ &\ \mathbf{imply}\ (\texttt{a2.incr\_spacing}\ \mathbf{\&\&}\ \texttt{a2.spacing\_done}))\end{aligned} \tag{8}$$

where *a3* is the agent which is in the *rdy_ch_lane* location, i.e., the agent has received a joining agreement, variable $joining\_vehicle.front$ indicates the identification of the preceding agent, flag $a2.incr\_spacing$ models that the preceding agent has received an "increase space" command from the leader and, finally, flag $a2.spacing\_done$ models whether agent *a2* has successfully increased its space. Flag $a2.spacing\_done$ turns true when $a2$ and *leader* are synchronised through $set\_spacing\_from$ channel, i.e., the leader is informed that $a2$ has successfully increased space. We can also verify this property for agents *a2* and *a4*. Property 8 is a safety requirement ensuring that a vehicle initiates "changing lane" only if sufficient spacing is provided.

The next property of interest is whether a joining request always ends up increasing space of the preceding vehicle. To express this property, we use the *leads to* property form, written $\varphi \rightsquigarrow \psi$. It states that whenever φ is satisfied, then eventually ψ will be satisfied. Such properties are written as $\varphi \dashrightarrow \psi$ in Uppaal. We verify the property (9) to show that whenever agent *a3* is in the *wait_j_agr* location, i.e., has sent a joining request, and agent *a2* is the preceding vehicle in the platoon, then eventually *a3* will receive an increasing space command and will perform the action.

$$\begin{aligned}&(\texttt{a3.waiting\_for\_j\_agr}\ \mathbf{\&\&}\ \texttt{l.joining\_vehicle.front} == 2)\\ &\dashrightarrow (\texttt{a2.incr\_spacing}\ \mathbf{\&\&}\ \texttt{a2.spacing\_done})\end{aligned} \tag{9}$$

To ensure that the spacing always decreases after a joining procedure, i.e., platoon returns back to a normal state, we verify that if ever the leader receives a joining request, it eventually sends a decreasing space command to the preceding agent unless the joined agent is the final one in the platoon.

$$\begin{aligned}&\mathbf{A}\Box\ ((\texttt{a3.join\_completed}\ \mathbf{\&\&}\ \texttt{l.joining\_vehicle.front} == 2)\\ &\ \mathbf{imply}\ (!\texttt{a2.incr\_spacing}\ \mathbf{\&\&}\ \texttt{a2.spacing\_done}))\end{aligned} \tag{10}$$

Given the required time for a vehicle to carry out "set spacing", "joining distance" and "changing lane" tasks, we are interested in verifying if an agent accomplishes joining the platoon within an expected interval: waiting time for agreement + changing lane + joining distance + waiting time for leader confirmation, represented in Property 11.

$$\begin{aligned}&\mathbf{A}\Box\ (\texttt{a2.join\_completed}\ \mathbf{imply}\\ &\ (\texttt{a2.process\_time} \geq 50\ \mathbf{\&\&}\ \texttt{a2.process\_time} \leq 90))\end{aligned} \tag{11}$$

Similarly, we check if an agent leaves a platoon within an expected interval: waiting time for agreement + changing lane + waiting time for leader confirmation. Waiting time for agreement is equal to the time needed to set space and waiting time for leader confirmation is zero because we assume switching steering controllers is immediate.

$$\begin{aligned}&\mathbf{A}\Box\ (\texttt{a2.leave\_completed}\ \mathbf{imply}\\ &\ (\texttt{a2.process\_time} \geq 30\ \mathbf{\&\&}\ \texttt{a2.process\_time} \leq 50))\end{aligned} \tag{12}$$

4.6. Validating the verification

Within our formal approach a range of abstractions are used, many being over-approximations. Practical simulation and testing is crucial, not only to validate the system being developed, but also to validate these abstractions. If, for example, we find a property that does not hold we must be sure that this is due to the system itself and not to a failure introduced through our abstraction process. Consequently, we provide a close link between formal verification, simulation and (increasingly) real-world testing. This is a general problem with all formal verification techniques applied to real-world systems. Since we can never precisely formalise the "real world" we must provide abstractions of the world and carry out verification with these. The abstractions must in turn be validated to see if they capture most of the viable real world behaviour. If they do not, then they will be refined and verification again carried out.

It must be emphasised that the agent code that we verify is actually the code that controls the vehicle, both in the TORCS simulation and in the real vehicle that we are developing. An imperfect environmental abstraction can mask possible problems in an agent model in a way which even formal verification of the agent including the environmental abstraction, fails to find. For instance, in the first attempt to model the platooning agent model, the environmental abstraction always delivered a change lane perception when a change lane action had been requested. This disguised the lack of required plans for the agent to deal with situations where a vehicle failed to change lane. The inappropriate abstraction was detected during simulation and provided means to refine the environmental abstraction and consequently the agent model. Thus, as long as the environmental abstractions are correct, we can be sure of the decisions made by the agent.

This clearly leaves the question of whether the environmental abstractions are appropriate or not. This is not something that can be handled by formal verification and so it is here that we utilise *validation*, both through TORCS simulation and physical vehicle tests. These help us assess how realistic and appropriate the abstractions we use are, and provide the impetus to refine these if necessary.

5. Concluding remarks

The verification of safety considerations for automotive platooning is a difficult task. There are several reasons for this.

- These are non-trivial hybrid autonomous systems, with each vehicle mixing feedback controllers and agent decision-making.
- There is a strong requirement to verify the *actual* code used in the implementation, rather than extracting a formal model of the program's behaviour — this leads on to *program* model-checking, which is resource intensive.
- Rational agents are essential for capturing high-level autonomy and, as there are no other practical systems able to model-check temporal and modal properties of complex BDI agents, we are led to AJPF.
- AJPF is *very* resource intensive (as we have seen, 12 hours for some agent properties) and cannot be practically used to verify whole system properties of automotive platooning.
- Especially when interacting with real vehicles we need to verify timed properties, as timing considerations can be crucial.

Thus it is perhaps not surprising that such formal verification has never been reported before. In order to address all of the above concerns, we have adopted a combined strategy. We use AJPF to verify individual agent properties, given realistic abstractions of environmental interactions. AJPF verification is viable for individual agents, though remains resource hungry. We then abstract from the BDI code and produce an abstract agent automaton suitable for use in Uppaal verification. We have shown how this can be carried out automatically, producing smaller timed automata with the "internals" of autonomous decision-making removed. Essentially, we remove the modelling of the decision-making process and just leave the possible decisions that can be made. This then allows us to formally verify platoon requirements and safety considerations, a sample of which we have included.

5.1. Related work

The California PATH project [35] has developed a design of platooning for intelligent highway systems. The design of manoeuvres which involve both discrete and continuous behaviour forms a large complex hybrid system. Safety verification of platooning has been discussed in [26,30] where vehicle dynamics is taken into account. A methodology for the design of safe hybrid controllers for automated vehicles has been presented based on game theory and optimal control techniques. The focus is on finding the safe continuous control laws for the leader and followers in different platoon manueuvers, e.g., join, split, lane change and leave. For instance, it is shown in [26] that if a follower finds itself closer than s from the preceding vehicle, a collision might likely happen. A hybrid I/O automaton approach has been applied to the platooning in [13] to describe and verify safety of merge manoeuvre for platooning. The purpose in [13] is to ensure that two adjacent platoons can safely combine to form a single platoon. The requirement for safe joining of two platoons is that the two platoons never collide at a relative velocity greater than a given bound. Sufficient conditions on the controller of the platoons are given to ensure the safety. The focus is on finding the optimal velocity that ensures the safety for merging two adjacent platoons. While a hybrid controller automata approach has been taken into account in these works for safety verification of controllers, the aim of our study is just to verify the high-level autonomous decision-making component of the hybrid system. Precisely, the hybrid automata approaches in [13,26,30] focus on multiple control requirements that the controller has to strive to satisfy and abstract the required coordination for different manueuvers; however, our approach addresses verification of the coordination for different manueuvers.

A spatial interval logic has been presented in [20,19] to abstract from the dynamics of vehicles to allow for spatial reasoning. The proposed logical framework aims to prove collision freedom of lane-change manoeuvres by introducing a local view of each vehicle. Similar to our approach, vehicle dynamics are separated from spatial reasoning; however, the focus, in [20,19], is on reasoning about controllers not the high-level decision making component of the system. A framework based on π-calculus has been presented in [5] for formal modelling of high-level decision making of platooning while the closed-loop control systems are modelled using hybrid automata. As with our approach, the discrete and continuous aspects of the system are separated, though no verification has been carried out in [5] to ensure the correctness of the high-level decision making component.

A combined verification approach for vehicle platooning is proposed in [6] where the system behaviour is specified via the CSP and B formal methods. The approach in [6] is based on refinement in which a system is developed in a stepwise manner. The vehicle behaviour is modelled as a B machine and the communication models as a CSP controller. The aim in [6] is to represent a correct model of a real physical vehicle for platooning while our approach aims to verifying the co-operation between vehicles in the automotive platoon and abstract the behaviour of a real physical vehicle. A compositional verification approach for vehicle platooning is introduced in [14] where feedback controllers and agent decision-making are mixed.

It should be noted that to the best of our knowledge, there is no directly comparable work in literature where the actual agent code used in the implementation has been verified.

5.2. Future work

There is clearly much future work to tackle. An obvious one is to continue efforts to improve the efficiency of AJPF.

Maintaining a safe platoon in case of *recoverable* latency and dissolving a platoon in the case of *unrecoverable* latency are two procedures that are not yet implemented in our verified agent code. Adding these two procedures to the agent grows the system space to the extent that AJPF fails to verify any property. Thus, we are investigating an agent abstraction at the level of goals, beliefs and intentions in order to use AJPF for verification of more complex agents.

Since we are concerned with certification of automotive platooning in practice, we are aiming to extract a more comprehensive list of formal properties from official platoon requirement documents. Related to this, we are also in the process of porting the agent architecture on to a real vehicle and so aim to continue and extend testing of the platooning algorithms in both physical, and simulation, contexts.

Finally, an important aspect of our two pronged strategy is to link the models used in Uppaal to the programs that AJPF uses. We provided an algorithm to achieve this but, in this paper we refined the Uppaal models by hand. Clearly, a next step is to fully automate this process.

Acknowledgements

This research was funded by EPSRC via the research project “Verifiable Autonomy”, at Liverpool (EP/L024845) and Sheffield (EP/L024942). Further details of the project are available at http://wordpress.csc.liv.ac.uk/va. We would like to kindly thank the reviewers for their detailed and valuable comments on our manuscript.

References

[1] R. Alur, C. Courcoubetis, D. Dill, Model-checking in dense real-time, Inf. Comput. 104 (1993) 2–34.
[2] G. Behrmann, A. David, K.G. Larsen, A tutorial on Uppaal, in: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Revised Lectures, in: Lect. Notes Comput. Sci., vol. 3185, Springer, 2004, pp. 200–236.
[3] C. Bergenhem, Q. Huang, A. Benmimoun, T. Robinson, Challenges of platooning on public motorways, in: Proc. 17th World Congress on Intelligent Transport Systems, 2010, pp. 1–12.
[4] M.S. Branicky, Introduction to hybrid systems, in: Handbook of Networked and Embedded Control Systems, Birkhäuser, 2005, pp. 91–116.
[5] J. Campbell, C.E. Tuncali, T.P. Pavlic, G. Fainekos, Modeling concurrency and reconfiguration in vehicular systems: a pi-calculus approach, in: Proc. 9th Interaction and Concurrency Experience Workshop of DicCoTec, ICE, 2016.
[6] S. Colin, A. Lanoix, O. Kouchnarenko, J. Souquières, Using CSP||b components: application to a platoon of vehicles, in: Formal Methods for Industrial Critical Systems, Springer, 2009, pp. 103–118.
[7] L.A. Dennis, Gwendolen Semantics, Technical report, Department of Computer Science, University of Liverpool, UK, 2017.
[8] L.A. Dennis, B. Farwer, Gwendolen: a BDI language for verifiable agents, in: AISB'08 Workshop on Logic and the Simulation of Interaction and Reasoning, AISB, 2008.
[9] L.A. Dennis, M. Fisher, N. Lincoln, A. Lisitsa, S.M. Veres, Reducing code complexity in hybrid control systems, in: Proc. 10th International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-Sairas), 2010.
[10] L.A. Dennis, M. Fisher, N.K. Lincoln, A. Lisitsa, S.M. Veres, Declarative abstractions for agent based hybrid control systems, in: Declarative Agent Languages and Technologies VIII, in: Lect. Notes Comput. Sci., vol. 6619, Springer, 2011, pp. 96–111.
[11] L.A. Dennis, M. Fisher, N.K. Lincoln, A. Lisitsa, S.M. Veres, Practical verification of decision-making in agent-based autonomous systems, Autom. Softw. Eng. 23 (3) (2016) 305–359.
[12] L.A. Dennis, M. Fisher, M.P. Webster, R.H. Bordini, Model checking agent programming languages, Autom. Softw. Eng. 19 (1) (2012) 5–63.
[13] E. Dolginova, N.A. Lynch, Safety verification for automated platoon maneuvers: a case study, in: Proc. International Workshop on Hybrid and Real-Time Systems, HART, 1997, pp. 154–170.
[14] M. El-Zaher, J.-M. Contet, P. Gruer, F. Gechter, A. Koukam, Compositional verification for reactive multi-agent systems applied to platoon non collision verification, Studia Inform. Universalis 10 (3) (2012) 119–141.

[15] M. Finger, D.M. Gabbay, Combining temporal logic systems, Notre Dame J. Form. Log. 37 (2) (1996) 204–232.
[16] M. Fisher, L.A. Dennis, M. Webster, Verifying autonomous systems, Commun. ACM 56 (9) (2013) 84–93.
[17] D. Gabbay, A. Kurucz, F. Wolter, M. Zakharyaschev, Many-Dimensional Modal Logics: Theory and Applications, Stud. Logic Found. Math., vol. 148, Elsevier Science, 2003.
[18] T.A. Henzinger, The theory of hybrid automata, in: Proc. 11th IEEE Symposium on Logic in Computer Science, IEEE Computer Society, 1996, p. 278.
[19] M. Hilscher, S. Linker, E.-R. Olderog, Proving safety of traffic manoeuvres on country roads, in: Theories of Programming and Formal Methods, in: Lect. Notes Comput. Sci., vol. 8051, Springer, 2013, pp. 196–212.
[20] M. Hilscher, S. Linker, E.-R. Olderog, A. Ravn, An abstract model for proving safety of multi-lane traffic manoeuvres, in: Proc. Int'l Conf. on Formal Engineering Methods, ICFEM, in: Lect. Notes Comput. Sci., vol. 6991, Springer-Verlag, 2011, pp. 404–419.
[21] J.E. Hopcroft, R. Motwani, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, 3rd edition, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
[22] L. Ilie, G. Navarro, S. Yu, On NFA Reductions, Springer, Berlin, Heidelberg, 2004, pp. 112–124.
[23] JPF… the Swiss Army Knife of Java^{TM} verification. http://babelfish.arc.nasa.gov/trac/jpf/, Accessed 26-January-2016.
[24] S. Konur, M. Fisher, S. Schewe, Combined model checking for temporal, probabilistic, and real-time logics, Theor. Comput. Sci. 503 (2013) 61–88.
[25] A. Kurucz, Combining modal logics, in: J. van Benthem, P. Blackburn, F. Wolter (Eds.), Handbook of Modal Logic, in: Stud. Log. Pract. Reason., vol. 3, Elsevier, 2007, pp. 869–924.
[26] J. Lygeros, D. Godbole, S. Sastry, Verified hybrid controllers for automated vehicles, IEEE Trans. Autom. Control 43 (4) (Apr. 1998) 522–539.
[27] O. McAree, S.M. Veres, Lateral control of vehicle platoons with on-board sensing and inter-vehicle communication, in: Proc. European Control Conference, ECC, 2016.
[28] R. Nair, M. Tambe, Hybrid BDI-pomdp framework for multiagent teaming, J. Artif. Intell. Res. 23 (2005) 367–420.
[29] Current State of EU Legislation- Cooperative Dynamic Formation of Platoons for Safe and Energy-optimized Goods Transportation, companion-project.eu/wp-content/uploads/COMPANION-D2.2-Current-state-of-the-EU-legislation.pdf; accessed 26 January 2016.
[30] A. Puri, P. Varaiya, Driving safely in smart cars, in: Proceedings of the 1995 American Control Conference, vol. 5, IEEE, 1995, pp. 3597–3599.
[31] A.S. Rao, Decision procedures for propositional linear-time belief-desire-intention logics, J. Log. Comput. 8 (3) (1998) 293–342.
[32] A.S. Rao, M.P. Georgeff, An abstract architecture for rational agents, in: Proc. 3rd International Conference on Principles of Knowledge Representation and Reasoning, KR, 1992, pp. 439–449.
[33] A.S. Rao, M.P. Georgeff, BDI agents: from theory to practice, in: Proc. 1st International Conference on Multi-Agent Systems, ICMAS, San Francisco, USA, 1995, pp. 312–319.
[34] SARTRE project, sartre-project.eu; accessed 26 January 2016.
[35] S.E. Shladover, PATH at 20 – history and major milestones, IEEE Trans. Intell. Transp. Syst. 8 (4) (2007) 584–592.
[36] D. Swaroop, String stability of interconnected systems: an application to platooning in automated highway systems, in: California Partners for Advanced Transit and Highways, PATH, 1997.
[37] Vienna convention on road traffic, http://www.unece.org/trans/conventn/crt1968e.pdf.
[38] W. Visser, K. Havelund, G. Brat, S. Park, F. Lerda, Model checking programs, Autom. Softw. Eng. 10 (2) (Apr. 2003) 203–232.
[39] M. Wille, M. Röwenstrunk, G. Debus, KONVOI: electronically coupled truck convoys, in: Human Factors for Assistance and Automation, 2008, pp. 243–256.
[40] M. Wooldridge, An Introduction to Multiagent Systems, John Wiley & Sons, 2002.
[41] M. Wooldridge, A. Rao (Eds.), Foundations of Rational Agency, Appl. Logic Ser., Kluwer Academic Publishers, Mar. 1999.