Safe and Optimal Adaptive Cruise Control

Kim Guldstrand Larsen(✉), Marius Mikučionis, and Jakob Haahr Taankvist

Department of Computer Science, Aalborg University,
Selma Lagerlöfs Vej 300, 9220 Aalborg Øst, Denmark
kgl@cs.aau.dk

Abstract. In a series of contributions Olderog et al. have formulated and verified safety controllers for a number of lane-maneuvers on multi-lane roads. Their work is characterized by great clarity and elegance partly due to the introduction of a special-purpose Multi-Lane Spatial Logic. In this paper, we want to illustrate the potential of current model-checking technology for automatic synthesis of optimal yet safe (collision-free) controllers. We demonstrate this potential on an Adaptive Cruise Control problem, being a small part of the overall safety problem considered by Olderog[1].

1 Introduction

These days the Google Self-Driving car is about to become a reality: legislation has been passed in several U.S. states allowing driverless cars, in April 2014, Google announced that their vehicles had been logging nearly 1.1 million km, and it is forecast that Google's self-driving cars will hit the roads this summer[1].

Also in Europe driverless cars have been actively pursued, both by the automotive industry itself and within a number of national and European research projects (e.g. FP7 and Horizon2020). With more and more traffic, European roads are becoming increasingly congested, polluted and unsafe. One potential solution to this growing problem is seen to be the use of small, automated, low-polluting vehicles for driverless transport in (and between) cities. Within the last decade, a number of European projects have been launched for making transport systems capable of fully automated driving, energy efficient and environmentally friendly while performing.

In addition, many individual driving assistant systems based on suitable sensors have been developed for cars. Moreover, car-to-car communication is considered to combine individual cars into more advanced assistance functionality. One particular class of such functionality considered by several researchers from the formal methods and verification community is that of driver assistance on multi-lane roads, with lane-changing and overtaking maneuvers. Overall this constitutes a hybrid systems verification problem, where the car dynamics, discrete

This paper is partially sponsored by the EU FET projects SENSATION and CASSTING as well as the Sino-Danish Basic Research Center IDEA4CPS.

[1] http://recode.net/2015/05/15/googles-homemade-self-driving-cars-are-hitting-the-roads-this-summer/

R. Meyer et al. (Eds.): Olderog-Festschrift, LNCS 9360, pp. 260–277, 2015.
DOI: 10.1007/978-3-319-23506-6_17

or timed controllers as well as road-specific assumptions should imply safety, i.e. absence of collisions.

The California PATH (Partners for Advanced Transit and Highways) project has spurred a series of research towards provable safe lane-maneuvers by developing advancing technologies that connect vehicles to surrounding infrastructure and other vehicles or automates vehicle processes[2]. For example the work by Lygos et al. [9,14], sketch a safety proof for manoeuvres of car platoons including lane change by taking car dynamics into account, but admitting safe collisions, i.e., collisions at a low speed.

More recently, within the German Transregional Collaborative Research Center AVACS – sponsored by the German Research Council (DFG) – Olderog together with colleagues made significant progress on proving safety of a number of automated maneuvers for overtaking on roads of varying complexity, ranging from highways (with uni-directional traffic) [11] to country roads (with two-directional traffic) [10]. The main contribution has been to show that (also here) one can separate the purely spatial reasoning from the underlying car dynamics in the safety proof. In particular, the approach taken by Olderog et. al. introduces a Multi-Lane Spatial Logic inspired by Moszkowski's interval temporal logic [16], Zhou, Hoare and Ravn's Duration Calculus [5] and Schäfer's Shape Calculus [17], and may be summarized by the following characteristics:

– The dynamics of cars are separated from the control laws – thus related to work by Raisch et al [15] and Van Schuppen et al [8].
– Development of the special-purpose multi-lane spatial logic MLSL to allow for easy formulation and verification.
– Design of controllers for lane-change maneuvers.
– Manual verification for proving safety of proposed controllers under general scenarios.

In addition to the Multi-Lane Spatial Logic (MLSL) [13], the formalism of timed automata [1] were used for specifying the protocol for safe lane-changing. However, though in principle possible, tool-support using a timed automata model-checker was never exploited in the above work. In this paper – celebrating the several contributions by Olderog to the area of the modeling and verification of real-time systems in general – we consider a small part of lane-change manoeuvres, namely the existence of a safe-distance controller (assumed in the above work of Olderog et al.). In particular, we aim at demonstrating how the most recent developments of the real-time model-checking tool UPPAAL [12] may be applied. Contrasting with the methodology of Olderog et.al. our method may be characterized as follows:

– Abstract away from dynamics for safety, but reconsidered dynamics for optimization (here to minimize the expected distance between cars).
– Use of a general purpose formalism in terms of various extensions of timed automata and games.

[2] http://www.path.berkeley.edu/

– Automatic synthesis of a range of controllers, ranging from safety controllers to optimal controllers, and to optimal yet safe controllers under specific scenarios.
– Extensive use of automated tool support, in terms of model checking, synthesis and optimization as provided by the most recent branch of the Uppaal tool suite, Uppaal Stratego [6,7].

The outline of the paper is as follows: in Section 2 we describe the Adaptive Cruise Control problem considered. In Section 3, we present the formalism of (weighted and stochastic) timed automata and games as used in Uppaal Stratego by means of a small Route Choosing Problem. Section 4 details our game model of the Adaptive Cruise Control problem, and Section 5 offers our results in terms of synthesis and analysis of safe, optimal and optimal-yet-safe controllers for the Adaptive Cruise Control problem.

Acknowledgement The authors would like to thank Anders P. Ravn for suggesting the Adaptive Cruise Control (or safety distance control) problem to us.

2 The Problem of Adaptive Cruise Control

We now define the case we will be analyzing using Uppaal Stratego in the remainder of the paper.

Two cars *Ego* and *Front* are driving on a road shown in Figure 1. We are capable of controlling *Ego*, but not *Front*. Both cars can drive a maximum of 20 m/s forward and a maximum of 10 m/s backwards. The cars have three different possible accelerations: $-2\,\mathrm{m/s^2}$, $0\,\mathrm{m/s^2}$ and $2\,\mathrm{m/s^2}$, between which they can switch instantly.

For the car to be safe there should be a distance of at least 5 m between them. Any distance less than 5 m between the cars is considered unsafe.

Ego's sensors can detect the position of *Front* only within 200 meters. If the distance between the cars is more than 200 meters then *Front* is considered to be *far away*. *Front* can reenter the scope of *Ego*'s sensor with arbitrary velocity it desires, as long as the velocity is smaller or equal to that of *Ego*.

We would then like to know the answers to the following questions:

– Is it possible that the cars crash?
  • If so, what is the probability of the cars crashing?
– Can we find a strategy for *Ego* such that the cars can never crash, no matter what *Front* does?

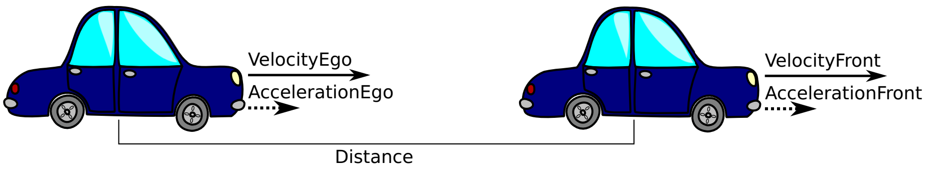


Fig. 1. Distance, velocity and acceleration between two cars.

    - Are there more than one of such strategies?
    - Which of the safe strategies lets *Ego* travel the furthest?
  - How does *Ego* respond to *Front*'s choices under all these different strategies?

3 Stochastic Priced Timed Games

For the synthesis of safe and optimal strategies, we will use (weighted and stochastic) timed automata and games, exploiting the tool UPPAAL STRATEGO [7] being a novel branch of the UPPAAL tool suite that allows to generate, optimize, compare and explore consequences and performance of strategies synthesized for stochastic priced timed games (SPTG) in a user-friendly manner. In particular, UPPAAL STRATEGO comes with an extended query language (see Table 1), where strategies are first class objects that may be constructed, compared, optimized and used when performing (statistical) model checking of a game under the constraints of a given synthesized strategy.

To illustrate the features of UPPAAL STRATEGO, let us look at the example in Fig. 2, providing an "extended" timed automata based model of a car, that needs to make it from its initial position `Start` to the final position `End`. In fact the model constitutes a timed *game*, where the driver of the car twice needs to make a decision as to whether (s)he wants to use a high road (`H1` and `H2`) or a low road (`L1` and `L2`). The four roads differ in their required travel-time (up to 100

Table 1. Various types of UPPAAL STRATEGO queries: "`strategy S =`" means strategy assignment and "`under S`" is strategy usage via strategy identifier `S`. Here the variables `NS`, `DS` and `SS` correspond to non-deterministic, deterministic and stochastic strategies respectively; `bound` is a bound expression on time or cost like `x<=100` and `n` is the number of simulations.

Strategy generators using [6]:	
Minimize objective:	`strategy DS = minE (expr) [bound]: <> prop`
Maximize objective:	`strategy DS = maxE (expr) [bound]: <> prop under NS`
Strategy generators using UPPAAL TIGA:	
Guarantee objective:	`strategy NS = control: A<> prop`
Guarantee objective:	`strategy NS = control: A[] prop`
Statistical Model Checking Queries:	
Hypothesis testing:	`Pr[bound](<> prop)>=0.1 under SS`
Evaluation:	`Pr[bound](<> prop) under SS`
Comparison:	`Pr[bound](<> prop1) under SS1 >= Pr[<=20](<> prop2) under SS2`
Expected value:	`value E[bound;n](min: prop) under SS`
Simulations	`simulate n [bound] { expr1, expr2 } under SS`
Symbolic model checking queries:	
Safety:	`A[] prop under NS`
Liveness:	`A<> prop under NS`
Infimum of value:	`inf { condition } : expression`
Supremum of value:	`sup { condition } : expression`

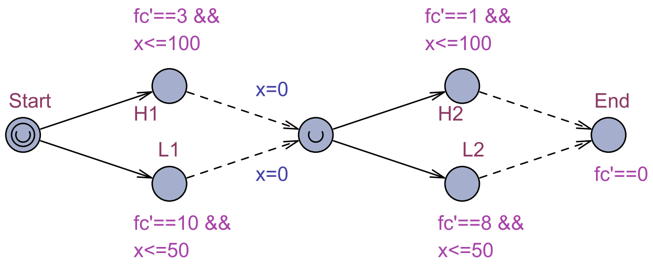
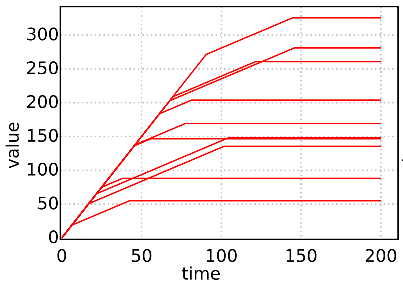


Fig. 2. The route choice problem for a car.

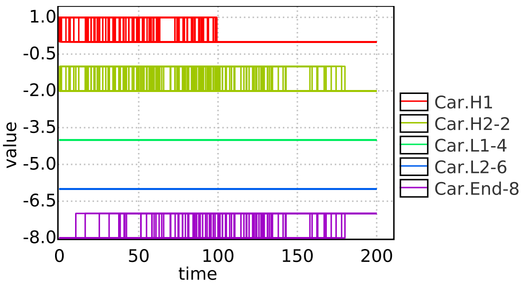
minutes respectively 50 minutes as reflected by the invariants on the clock `x`). Also the roads differ in fuel-consumption reflected by the difference in the rate of the continuous variable `fc` (representing the total amount of fuel consumed). Whereas the choice of road is up to the driver of the car to control (indicated by the solid transitions), the actual travel-time of the road is uncontrollable (indicated by the dashed transitions) reflecting the uncertainty of the amount of traffic on the particular day. In one scenario, the objective of the car it to choose the combination of roads that will ensure the shortest overall travel-time even in the most hostile traffic situation on the four roads. Under this interpretation, Fig. 2 represents a timed game. However, it may also be seen as a stochastic priced timed game (SPTG), assuming that the travel-times of the four roads are chosen by uniform distributions, and the objective of the control strategy is to minimize the expected overall travel-time, or the expected overall fuel-consumption (e.g. the rate or fuel-consumption `fc'==3` on the first high road `H1` indicates that the cost variable `fc` grows with rate 3 in this location).

We are interested in synthesizing strategies for various objectives. Being primarily concerned with fuel-consumption, the query

```
strategy Opt = minE (fc) [<=200] : <> Car.End
```



(a) `fc` trajectory samples. Fuel consumption on the vertical axis



(b) Road choice samples.

Fig. 3. Evaluation of strategy `Opt` via simulation.

will provide (by reinforcement learning[3]) the strategy `Opt`, that minimizes the expected total fuel-consumption, learning from runs which are maximally 200 time units long. The relativized query `E[<=200 ; 1000] (max: fc) under Opt`, generates 1000 runs of length 200 time units and then averages the maximum value of `fc` from each run. this is used to estimate the expected cost to be `200.39`. Figure 3a summarizes 10 random runs according `Opt` illustrating fuel-consumption. None of the runs had a fuel consumption of 400 indicating that we always choose the energy-efficient roads. In Figure 3b we see that this is actually the case as the simulations always choose to go to locations `H1` and `H2`, which models the energy-efficient roads.

Now, assume that the task *must* be completed before 150 time-units. From Fig. 3 it can be seen that the strategy `Opt` unfortunately does not guarantee this, as there are a few runs which exceeds 150 before reaching `End`. However, the query

```
strategy Safe = control: A<> Car.End and time<=150
```

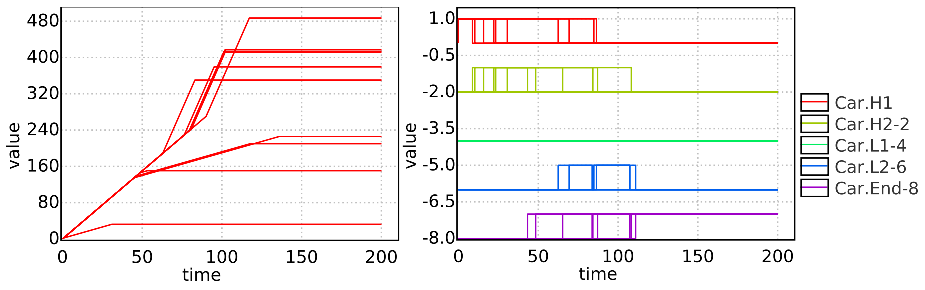
will generate the most permissive (non-deterministic) strategy `Safe` that guarantees this bound but unfortunately with a high expected total fuel-consumption of `342.19`. However, the relativized learning query

```
strategy OptSafe = minE (fc) [<=200] : <> Car.End under Safe
```

will provide a sub-strategy `OptSafe` that minimizes the expected total fuel-consumption – here found to be `279.87` – subject to the constraints of `Safe`. Figure 4 summarizes 10 random runs according to `SafeOpt`, incidating that only road `L1` is never choosen. Also, the failed model checking of `E<> Car.H2 and time>=51 and Car.x==0 under Safe` reveals that the high road `H2` may only be choosen in case the first phase is completed before 50 time-units, confirming the observations from the simulations.

In general, as shown in the overview Fig. 5, Uppaal Stratego will start from a SPTG $\mathcal{P}$. It can then abstract $\mathcal{P}$ into a timed game (TGA) $\mathcal{G}$ by simply ignoring prices and stochasticity in the model. Using $\mathcal{G}$, Uppaal Tiga [2] may now be used to (symbolically) synthesize a (most permissive) strategy σ meeting a required safety or (time-bounded) liveness constraint ϕ. The TGA $\mathcal{G}$ under σ (denoted $\mathcal{G}|\sigma$) may now be subject to additional (statistical) model checking using classical Uppaal [3] and Uppaal SMC [4]. Similarly, the original STGA $\mathcal{P}$ under σ may be subject to statistical model checking. Now using reinforcement learning[6], we may synthesize near-optimal strategies that minimizes (maximizes) the expectation of a given cost-expression *cost*. In case the learning is performed from $\mathcal{P}|\sigma$, we obtain a sub-strategy σ^o of σ that optimizes the expected value of *cost* subject to the hard constraints guaranteed by σ. Finally, given σ^o, one may perform additional statistical model checking of $\mathcal{P}|\sigma^o$.

[3] The reinforcement learning uses machine learning techniques to learn strategies from sets of randomly generated runs. See [6] for more details.



(a) `fc` trajectory samples. (b) Road choice samples.

Fig. 4. Evaluation of strategy `OptSafe` via simulation.

$$\begin{tikzcd}
\overset{\text{timed game}}{\mathcal{G}} \arrow[rr, "\text{(synthesis)}", "\phi"'] & & \overset{\text{strategy}}{\sigma} \arrow[r, "\text{time automata ((statistical)model checking)}"] \arrow[dl] & \mathcal{G}|\sigma \\
\underset{\text{stochastic priced timed game}}{\mathcal{P}} \arrow[u, "\text{abstraction}"] \arrow[r, dashed] & \mathcal{P}|\sigma \arrow[r, "\min E(C)", "\text{(optimization) Learning}"'] & \underset{\text{strategy}}{\sigma^\circ} \arrow[r, "\text{stochastic timed automata (statistical model checking)}"'] & \mathcal{P}|\sigma^\circ
\end{tikzcd}$$

Fig. 5. Overview of Uppaal Stratego

4 Modeling the Adaptive Cruise Control

In this section we introduce a model of the scenario described in Section 2.

We model the velocities and accelerations of the cars, where both cars can choose the acceleration they wish to have in each second. There are two sets of variables modeling the velocity and distance between the cars. The first set is a discrete set of variables. These variables are updated each second using the method `updateDiscrete()` seen in Listing 1. Note that when `distance > maxSensorDistance` we keep the distance constant, as the sensor cannot read how far away *Front* is anymore.

The second set measures the velocity and distance in a continuous way. The equations in Listing 2 define the rate of growth for continuous variables, we see that these are updated as expected. The rate of `rDistance` is defined using the function `distanceRate`, this function is presented in Listing 3.

What we see in Listing 3 is that when *Front* is *far away* from *Ego* we keep the distances between the cars constantly at the `maxSensorDistance` by setting the rate of `rDistance` to zero like with the discrete distance. If *Front* is not *far away* the rate of the distance is the expected `velFront - velEgo`, thus the relative velocity of the cars.

```
void updateDiscrete(){
    int oldVel, newVel;
    oldVel = velocityFront - velocityEgo;
    velocityEgo = velocityEgo + accelerationEgo;
    velocityFront = velocityFront + accelerationFront;
    newVel = velocityFront - velocityEgo;
    if (distance > maxSensorDistance) {
        distance = maxSensorDistance + 1;
    } else {
        distance += oldVel + (newVel - oldVel)/2;
    }
}
```

Listing 1. The code updating the discrete variables: `updateDiscrete()`.

```
rVelocityEgo' == accelerationEgo && rVelocityFront' ==
accelerationFront && rDistance' ==
distanceRate(rVelocityFront,rVelocityEgo, rDistance) &&
   D' ==
rDistance
```

Listing 2. The differential equations controlling the continuous variables.

Figure 6 validates by simulation that the discrete `distance` is correctly tracking the continuous trajectory of `rDistance`: the updated discrete values lie exactly on the continuous curve; the same applies to the difference of velocities.

Four parallel automata model and control the scenario: the system component, the models of *Ego* and *Front*, and the monitor component. The system component controls the system and enforces the discretisation of time, the monitor component monitors the state of the system and updates the hybrid variables correspondingly. We do not show the monitor component here as it is simply one location with the equations in Listing 2 as its invariant.

```
double distanceRate(double velFront, double velEgo,
   double dist) {
    if (dist > maxSensorDistance) return 0.0;
    else return velFront - velEgo;
}
```

Listing 3. The function `distanceRate()` defining how `rDistance` grows.

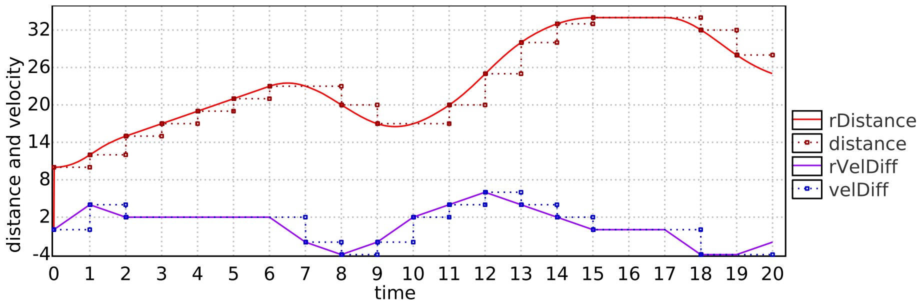


Fig. 6. Continuous and discrete distance (above) and velocity (below).

4.1 System Component

The *System* component, seen in Fig. 7 controls the discretisation of time, and also makes sure that *Ego* chooses its acceleration before *Front*. This means that when *Ego* chooses its acceleration for the next second it does not know what the acceleration of *Front* will be in the next step. It is also responsible for updating the discrete variables via the function `updateVelocities()`.

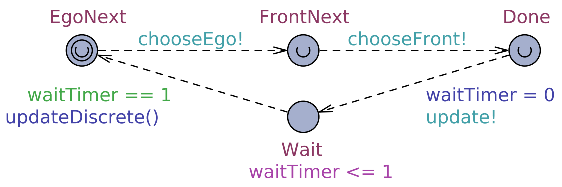


Fig. 7. The system component of the model: the component first lets *Ego* choose an acceleration, and then *Front*, i.e. *Ego* chooses an acceleration without knowing which acceleration *Front* will choose. The accelerations are chosen at discrete time instances every second.

Note that the only location in the *System* component which is not urgent (no time can pass in an urgent location) is the `Wait` location. Thus the cars have to choose their acceleration urgently.

When the system component has finished a loop and is in the `UpdateFront` location it will send an `update` broadcast before going to the `Wait` location, this signal is used by *Front* to determine weather it is *far away*.

4.2 The Model of *Ego* and *Front*

The two cars are modeled with a component each. The basic models of the cars are the same, however *Front* has extra behavior for the case when it is *far away*. We describe the model of *Ego* and in the end we describe the differences in the component for *Front*.

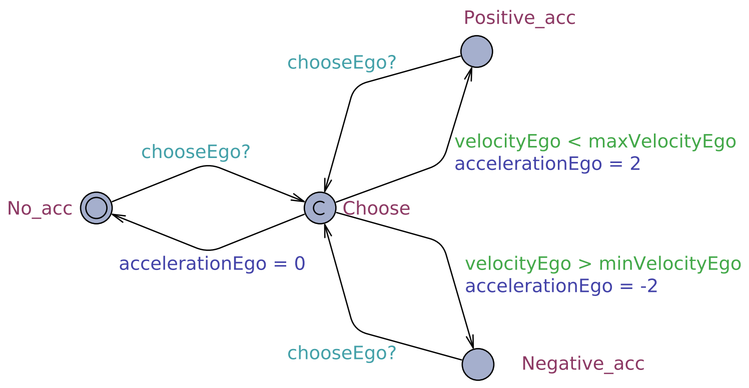


Fig. 8. Model of *Ego*.

The component for *Ego* can be seen in Fig. 8. There are four locations, the three locations which are not committed[4]; `No_acc`, `Positive_acc` and `Negative_acc` denotes the choice of acceleration in the current second, the last, `choice` is the location from which the choice is made. We also keep track of the acceleration via the variable `accelerationEgo`.

When the car gets a signal `chooseEgo?` it will go to the committed location `Choose`. From this location *Ego* will choose which acceleration to have in the next round. The location is committed to make sure *Ego* chooses it's acceleration before *Front*.

In this component we also enforce that the car cannot go faster than it's top speed (both backwards and forwards). This is done via the guards `velocityEgo < maxVelocityEgo` and `velocityEgo > minVelocityEgo` which ensure that it is only possible to have a positive acceleration if the current velocity is less than the top speed, and similarly it is not possible to have a negative acceleration if the current velocity is greater than the maximum negative speed.

The model of *Front* can be seen in Fig. 9. We see that it has the same set of places as *Ego* and the same set of variables. In addition to these it has the location `Faraway` and two committed locations. *Front* will go to the location `Faraway` if the distance between the cars is greater than `maxSensorDistance`, and it gets the `update` signal from the `System` component. The next time it gets the `chooseFront` signal there is fifty percent chance it will stay *far away* and fifty percent chance it will reenter the sensor distance. If it reenters the sensor distance it will choose which velocity it wants via the select statement `i:int[minVelocityFront, maxVelocityFront]`.

[4] When a component is in a committed location, the next transition *must* be from this location, without any delay.

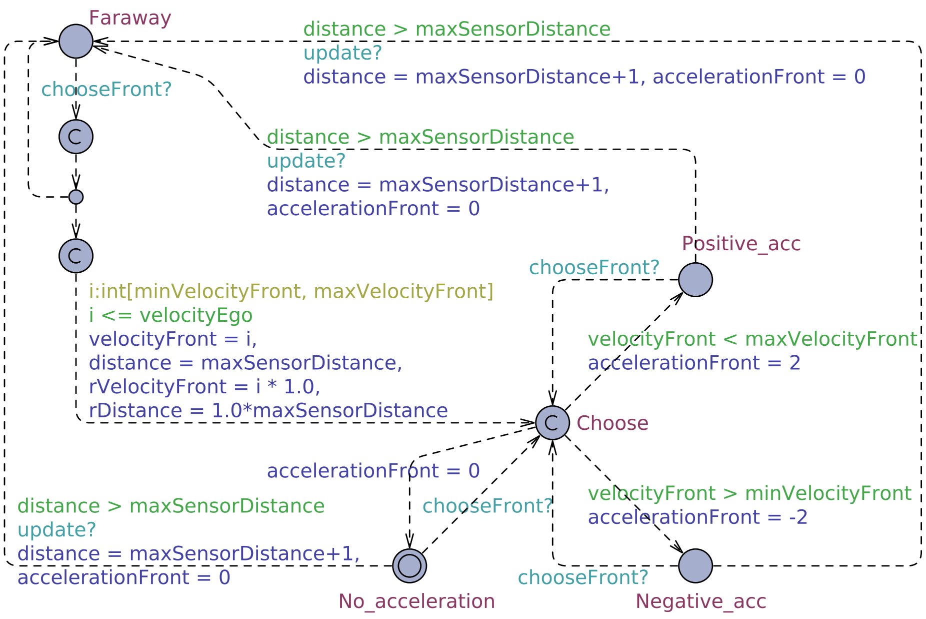

Faraway
distance > maxSensorDistance
update?
distance = maxSensorDistance+1, accelerationFront = 0
chooseFront?
distance > maxSensorDistance
update?
distance = maxSensorDistance+1,
accelerationFront = 0
Positive_acc
chooseFront?
i:int[minVelocityFront, maxVelocityFront]
i <= velocityEgo
velocityFront = i,
distance = maxSensorDistance,
rVelocityFront = i * 1.0,
rDistance = 1.0*maxSensorDistance
velocityFront < maxVelocityFront
accelerationFront = 2
Choose
accelerationFront = 0
distance > maxSensorDistance
update?
distance = maxSensorDistance+1,
accelerationFront = 0
chooseFront?
velocityFront > minVelocityFront
accelerationFront = -2
chooseFront?
No_acceleration
Negative_acc

Fig. 9. Model of *Front*.

5 Synthesis and Analysis

With the model defined we can start analyzing it. UPPAAL STRATEGO offers a set of different analysis tools which view the model in different ways.

UPPAAL STRATEGO can view the model as a game with a number of different players.

Half player game In a half player game all choices are stochastic. The part of UPPAAL STRATEGO concerned with this kind of games is UPPAAL SMC. In half player games the kind of questions asked is typically, *what is the probability to reach this state?*. In UPPAAL STRATEGO there has to be a timebound, so the question is then, *what is the probability of reaching this state in t time units?*.

One player game In one player games all choices are nondeterministic. The part of UPPAAL STRATEGO which is concerned with this kind of games is traditional UPPAAL. In one player games we ask questions such as *is it possible to reach this state?*.

One and a half player game In one and a half player games some choices are nondeterministic and others are stochastic. UPPAAL STRATEGO is the first version of UPPAAL to include support for one and a half player games. A question asked in a one and a half player game could be *what is the best strategy to maximize the expected value of x?*. As with half player games there has to be a timebound in Uppaal Stratego, so the question will be *what is the best strategy to maximize the expected value of x in t time units?*

Two player game In two player games all choices are nondeterministic, one set of choices are made by the controller, the other set is made by the environment. The part of Uppaal which handles two player games is Uppaal Tiga. Questions in two player games could be *does there exist a strategy that makes sure I always reach this state, no matter which choices the environment makes?*

5.1 No Strategy

We can now analyze the model described in Section 4 using Uppaal Stratego. The first question we could ask is if the cars will always be at least five meter from each other no matter what happens. This is a one player question. We can use the following Uppaal query to answer the question:

```
A[] distance > 5
```

This is not true and Uppaal gives us a trace which tells us that if *Ego* starts by accelerating forward and *Front* starts by accelerating backwards then the cars will collide.

A next question could then be what the probability that the distance between the two cars is less than five after 100 seconds. This is a half player question and is answered using the Uppaal SMC query:

```
Pr[<=100] (<> distance <= 5)
```

This tells us that the probability of a crash if both cars drive randomly is in the interval $[0.856, 0.866]$ with a confidence of 95%. In Fig. 10 we can see that the probability of a crash is greatest in the beginning of the run and then decreases. We can also see that the average crash happens around 20 seconds after the cars starts moving.

We can also do a set of random simulations of the model to get a more nuanced view on how the model behaves. A set of simulations can be seen in Fig. 11, on the horizontal axis we see the time and on the vertical we see the distance between the cars.

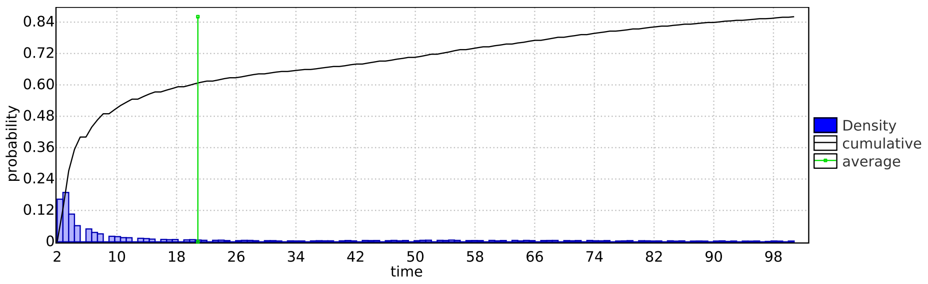


Fig. 10. The cumulative probability and the probability density of a crash.

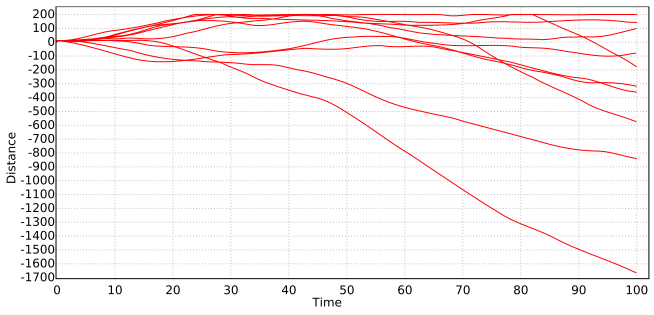


Fig. 11. A set of ten random simulations over the half player version of the model.

5.2 Safe Strategy

Clearly a probability of 85% for a crash is not acceptable. To remove the risk of a crash we consider the model as a two player game where we control *Ego* and the antagonistic environment controls *Front*. We can then ask for a strategy which makes sure that no matter what *Front* does we will avoid a crash. We can request such a strategy using the query:

```
control: A[] distance > 5
```

With UPPAAL STRATEGO it is possible to save a strategy in a named variable, we will do this as we would like to use the strategy later. The query then looks like this:

```
strategy safe = control: A[] distance > 5
```

This will give us the most permissive strategy which makes sure that the distance between the cars is kept greater than 5. The most permissive strategy is a strategy which in every state suggests the biggest set of actions possible to the controller. We can now do a set of simulations under this strategy. This is done using the query:

```
simulate 10 [<=100] rDistance under safe
```

This gives us the plot in Fig. 12. What we can see is that here, as opposed to Fig. 11 where there was no strategy, the distances is always greater than five for all ten runs, to be absolutely sure this is always the case we use the query:

```
A[] distance > 5 under safe
```

Which is of course true as the strategy is constructed to ensure exactly that, in the same way we can verify that it is possible to have a distance of 6.

Another one player query we can ask is:

```
inf{velocityFront-velocityEgo==v}: distance under safe
```

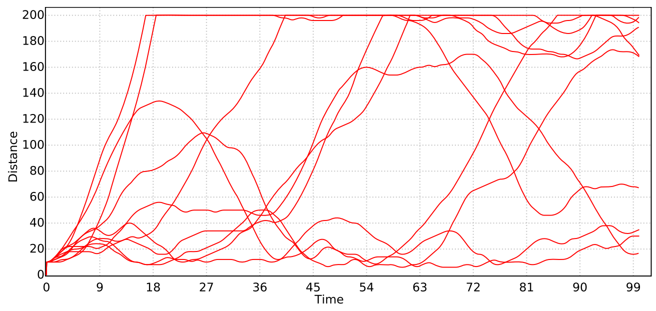


Fig. 12. A set of ten random simulations over the half player version of the model subject to a strategy which guarantees that distance is greater than five.

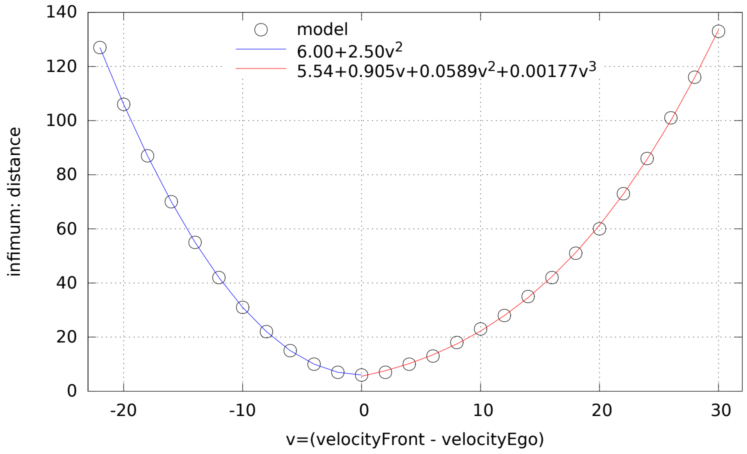


Fig. 13. Smallest distance possible under the `safe` strategy as a function of speed difference computed using `inf{velocityFront-velocityEgo==v}: distance under safe` for each v value. Connecting lines are from linear regression analysis.

this will return the smallest (inf) distance seen in any state where

`velocityFront-velocityEgo==v`,

thus the smallest distance at the relative velocity `v` possible under the safe strategy. In Fig. 13 we see the result of running this query with v at different values. When the velocity difference v is negative, the safe distance dependency is exactly quadratic: this is the distance the *Front* car would cover towards *Ego* if it kept accelerating and *Ego* were not able to keep up. Thus the faster the cars are driving towards each other the greater the distance should be to ensure the cars are safe. When v is positive the safe distance slope is not as steep, but it follows a similar worst case scenario that the *Front* may immediately start accelerating towards *Ego*, except it will take longer to make the velocity difference negative.

5.3 Safe and Fast Strategy

In Section 5.2 we generated the most permissive strategy which made sure the cars was safe. However the most permissive strategy is the union of all strategies, thus we have the opportunity of choosing the best strategy according to some measure. In Listing 2 we see that the rate of `D` is defined as `D' == rDistance`. This means that the value of `D` is the accumulated distance during the run. We can choose a strategy which minimizes `D`, this will mean that *Ego* will try to stay close to *Front*, but it will still stay far enough away that it will be safe. We can learn such a strategy using the query

```
strategy safeFast = minE (D) [<=100]: <> time >= 100 under safe
```

This is a one and a half player query, thus we assume the environment to be stochastic. We can then learn a strategy which attempts to minimize the expected value of `D`, as done in the query.

Using the learned strategy `safeFast` we can then make a set of simulations with `simulate 10 [<=100] rDistance under safeFast`, this gives us the plot in Fig. 14.

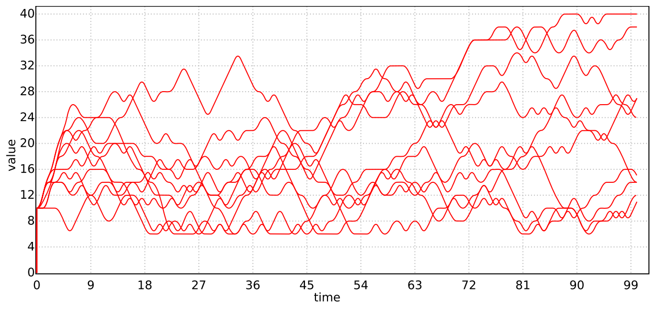


Fig. 14. A set of ten random simulations over the half player version of the model subjected to the strategy `safeFast` which optimizes the value of `D`, while still guaranteeing that the distance is greater than five.

We see that under this strategy the distance between the cars are much smaller, thus we now have a strategy that is not only safe but which also attempts to make the distance between the cars small.

We can also directly compare the two strategies `safe` and `safeFast` using the query:

```
Pr[rDistance <= maxSensorDistance + 1] <> time >= 100 under safe.
```

This query asks what the probability of reaching `time >= 100` is in runs which are bounded by `rDistance`. As `rDistance` never is greater than maxSensorDistance, we know that all runs will reach `time >= 100`. Thus the query reports

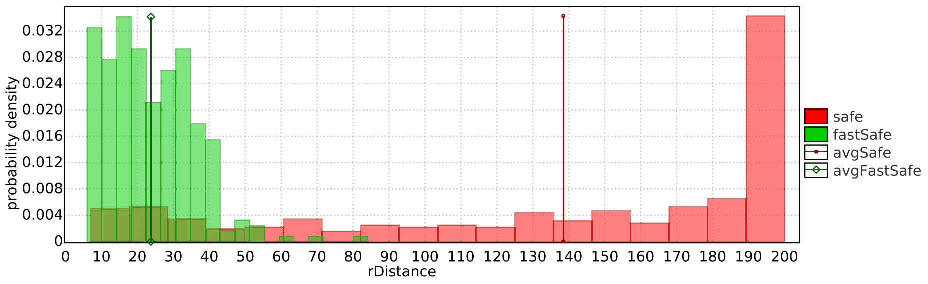
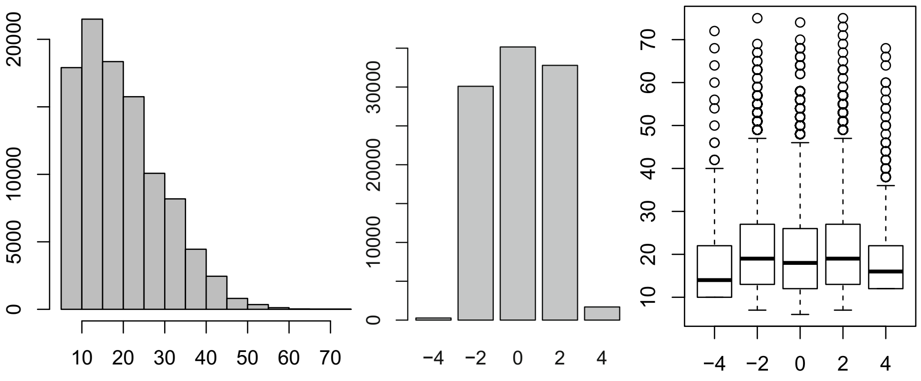


Fig. 15. The probability density distribution over `rDistance` at `time >= 100` thus after 100 time units under the strategies `safe` and `safeFast`. The (dark) red bars for `safe` and the (light) green bars for `safeFast`.



(a) Distance histogram. (b) Velocity histogram. (c) Distance over velocity.

Fig. 16. Statistics from 1000 fixed-time-step simulations under `safeFast` strategy: overall 100000 data points about distance and velocity difference.

correctly that the probability is 1. However we can then look at what `rDistance` was when we reached `time >= 100`.

In Fig. 15 we see the probability density distribution for the value of `rDistance` when `time >= 100` generated using the query above. The red part shows the distribution under `safe` and the green part shows the distribution under `safeFast`.

What we can see is that under `safe` *Ego* just stays far away from *Front*, as this is safe. This is of cause boring as *Ego* will then never really move forward. Under `safeFast` on the other hand we see that *Ego* stays relatively close to *Front* thus minimizing `D` just like we intended.

Figure 16 summarizes the characteristics of the `safeFast` strategy of individual simulation points. The histograms show that the distance is limited by 70 metres and most of the time the cars are close to each other. Interestingly the velocity is also limited to a narrow range meaning that *Ego* manages to mimic the speed of *Front* and only rarely the speed difference greater than 2m/s.

6 Conclusion

We have demonstrated that safe and optimal distance controllers may be synthesized automatically using the recently emerged Uppaal Stratego branch of the Uppaal tool suite. What remains for future work that we would like to undertake, is to synthesize the complete safe overtaking protocol suggested by Olderog et al in their previous work.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994)
2. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K.G., Lime, D.: UPPAAL-tiga: time for playing games!. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 121–125. Springer, Heidelberg (2007)
3. Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: Uppaal 4.0. In: Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems, QEST 2006, pp. 125–126. IEEE Computer Society, Washington (2006)
4. Bulychev, P.E., David, A., Larsen, K.G., Mikučionis, M., Poulsen, D.B., Legay, A., Wang, Z.: UPPAAL-SMC: statistical model checking for priced timed automata. In: Wiklicky, H., Massink, M. (eds.) Proceedings 10th Workshop on Quantitative Aspects of Programming Languages and Systems, QAPL 2012. EPTCS, Tallinn, Estonia, vol. 85, pp. 1–16, March 2012
5. Chaochen, Z., Hoare, C.A.R., Ravn, A.P.: A calculus of durations. Information Processing Letters **40**(5), 269–276 (1991)
6. David, A., Jensen, P.G., Larsen, K.G., Legay, A., Lime, D., Sørensen, M.G., Taankvist, J.H.: On time with minimal expected cost!. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 129–145. Springer, Heidelberg (2014)
7. David, A., Jensen, P.G., Larsen, K.G., Mikučionis, M., Taankvist, J.H.: Uppaal Stratego. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 206–211. Springer, Heidelberg (2015)
8. Habets, L.C.G.J.M., Collins, P.J., van Schuppen, J.H.: Reachability and control synthesis for piecewise-affine hybrid systems on simplices. IEEE Transactions on Automatic Control **51**(6), 938–948 (2006)
9. Haddon, J.A., Godbole, D.N., Deshpande, A., Lygeros, J.: Verification of hybrid systems: monotonicity in the AHS control system. In: Alur, R., Sontag, E.D., Henzinger, T.A. (eds.) HS 1995. LNCS, vol. 1066. Springer, Heidelberg (1996)
10. Hilscher, M., Linker, S., Olderog, E.-R.: Proving safety of traffic manoeuvres on country roads. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) Theories of Programming and Formal Methods. LNCS, vol. 8051, pp. 196–212. Springer, Heidelberg (2013)
11. Hilscher, M., Linker, S., Olderog, E.-R., Ravn, A.P.: An abstract model for proving safety of multi-lane traffic manoeuvres. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 404–419. Springer, Heidelberg (2011)
12. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. International Journal on Software Tools for Technology Transfer **1**(1–2), 134–152 (1997)
13. Linker, S., Hilscher, M.: Proof theory of a multi-lane spatial logic. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) ICTAC 2013. LNCS, vol. 8049, pp. 231–248. Springer, Heidelberg (2013)

14. Lygeros, J., Pappas, G.J., Sastry, S.: An approach to the verification of the center-TRACON automation system. In: Henzinger, T.A., Sastry, S.S. (eds.) HSCC 1998. LNCS, vol. 1386, pp. 289–304. Springer, Heidelberg (1998)
15. Moor, T., Raisch, J., O'Young, S.: Discrete supervisory control of hybrid systems based on l-complete approximations. Discrete Event Dynamic Systems **12**(1), 83–107 (2002)
16. Moszkowski, B.: A temporal logic for multilevel reasoning about hardware. Computer **18**(2), 10–19 (1985)
17. Schäfer, A.: A calculus for shapes in time and space. In: Liu, Z., Araki, K. (eds.) ICTAC 2004. LNCS, vol. 3407, pp. 463–477. Springer, Heidelberg (2005)