2014 IEEE Intelligent Vehicles Symposium (IV)
June 8-11, 2014. Dearborn, Michigan, USA

Testing Autonomous Driving Systems against Sensor and Actuator Error Combinations

Pascal Minnerup[1] and Alois Knoll[2]

***Abstract*— Simulation as a tool for validating the behavior of an autonomous driving system can include sensor and actuator errors. A combination of such errors might lead to undesired behavior not occurring with a single error. In this paper we present a new concept for efficiently covering these error combinations. We propose to divide the simulation sequence into small separately simulated pieces that can be assembled to cover more patterns of error combinations. This way, parts of a simulation sequence that are similar for multiple error patterns are run only once. Our experimental results show that the concept can find weaknesses of planning and control systems and is faster than running a separate simulation sequence for each error pattern.**

I. Introduction

Simulation can help to evaluate planning and control systems without executing them on the actual car. In a simulation environment, the developer has full control over physical failures, can start and stop the test at any time and can even pause for debugging directly in the source code. This power can be used to validate planning and control systems against all expected errors before launching cost intensive tests on the actual vehicle. Simulation can also cover error patterns that are relatively unlikely but relevant for mass-production cars. Such patterns are difficult to cover using tests on the physical vehicle.

However, an exhaustive test in the simulation would also require running the simulation for each combination of errors. Plus, the combination of errors can happen at any point in time. This would make the number of simulation runs grow exponentially with the length of the simulation. Furthermore, testing each error pattern separately wastes time by running some parts of the simulation many times although they are very similar. This problem is depicted in Figure 1. Some parts of the simulation can be nearly equal for different error patterns. In Figure 1, the upper arrow represents a simulation run without errors. The lower arrow shows a simulation run with a position error occurring after some time. The part of the simulation until the first error happens would be exactly equal ("same start"). Sometime after the error happens, the controller will have minimized the position error and thus the simulation will continue almost equally for both error patterns (compare "same end" in Figure 1). In this paper, an approach is presented that takes advantage of these common simulation parts in order to get a similarly effective result as simulating exponentially more combinations of errors.

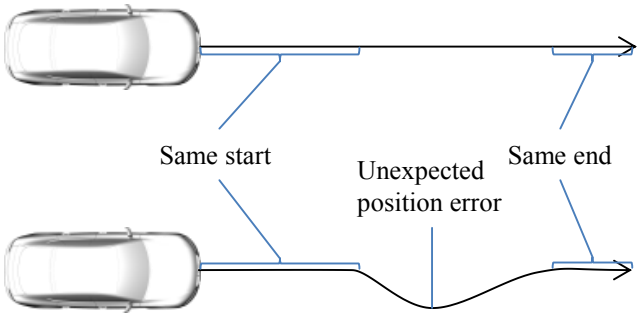


Fig. 1: Difference of driven paths with two different error patters: no error and a position error; For covering both paths, our concept simulated the common start and end only once.

II. Related work

While many research teams – in particular of the DARPA Urban Challenge – highlight the importance of a simulation environment [1][2][3][4][5], most teams do not focus on the simulation strategy. In [1], the authors describe a simulation environment that can be defined by a scene file and changed by a graphical user interface. The authors of [2] see testing as a "cornerstone of the development process" and use a simulation to test capabilities required for the Urban Challenge. Team VictorTango [3] agrees on the importance of simulation and testing. Their simulation can be configured by static scene files, too. The Skynet team [4] shows results of Urban Challenge simulations. The results do not mention large set of tested combinations; hence the simulation was probably done linearly. The authors of [5] describe how to adjust the architecture such that software in the loop tests are possible. Using their systems, they perform long linear time test runs. None of the teams mentions systematically introducing sensor and actuator errors to their tests or otherwise systematically testing variations of the scenarios.

In industry "Model-in-the-Loop" (MIL) and "Hardware-in-the-Loop" (HIL) tests are performed. The authors of [6] describe MIL tests and present a new method for performing HIL tests. The goal of these tests is to simulate a physical test drive as good as possible. The tested scenarios are safety critical maneuvers including faults. Sensor errors other than complete failures are not systematically tested.

Systematic testing has been researched by other groups. The authors of [7][8] present a method for generating test

[1]Pascal Minnerup is with fortiss GmbH, affiliated institute of Technische Universität München, Munich, Germany minnerup@fortiss.org
[2]Alois Knoll is with Robotics and Embedded Systems, Technische Universität München, Munich, Germany knoll@in.tum.de

cases by evaluating the results of each test run. Using a heuristic they try to push the test executions towards bad events. For them, the simulation is only altered by different starting constellations like the shape of the obstacles. In [9] a different approach for finding such test cases is used: symbolic execution. This way, the authors virtually execute a program using all possible combinations of input values. The approach presented in our paper also tests combinations of events and additionally takes advantage of the geometric structure of path planning and control in order to increase the efficiency of the computation. Additionally, the research group of [10] focuses on parallelizing the simulation flows. The approach presented in our paper is designed for straight forward parallelization, as each simulated step can be computed on a separate computer.

Using better tools for finding rare undesired behavior also allows focusing more on the average case quality of a planning and control system. For this challenge, the authors of [11] advocate a design procedure for controllers that focuses on both, average case quality and worst case reliability. In our paper, we support the idea that controller design should focus more on average case quality. The analysis presented in this paper can be used for exploring the worst case events. It can also be used after a controller upgrade as regarded in [12]. After such a controller upgrade, the software engineers can rerun the reliability analyses.

An alternative to simulation are formal approaches as described by the author of [13]. His concept computes reachable states of a system with the example of an autonomous car. The dissertation uses a simplified model of the car with some parameters being given as a probability distribution. For this model, he can prove that certain states are not reachable. The disadvantage of such a verification technique is that it checks an abstraction of the system rather than the software system itself and due to an over approximation it is not able to show counter examples. Furthermore, the approach shown in our paper can be extended to a larger set of possible errors with little effort.

III. Branching and Merging Simulation States

As explained in the previous section, in most research groups, simulation scenarios are run one by one. Figure 2 shows a possible simulation scenario. The car starts with some *Start* configuration and is ordered to park, i.e. move to a *Goal* position while avoiding the *Obstacles*. The long curve is the path the planning layer produces in this simulation. In the example, the simulation takes five time steps. The captions (0t-5t) indicate the positions of the car at different points in time.

In this paper, we divide the simulation into small time intervals. Figure 3 shows the first interval of the simulation. Additionally there is a red line indicating the trajectory followed by the car with an error occurring: a strong drift of the position estimation. If the position estimation has a drift in one direction, the controller will try to move the car into the opposite direction. Hence the end of the red line is

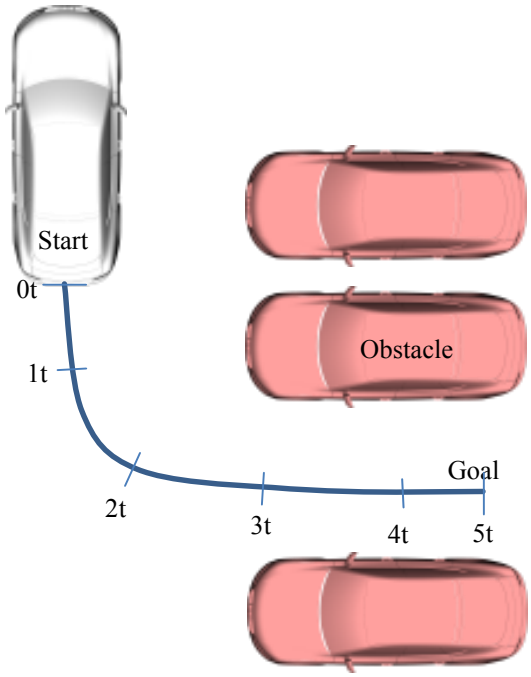


Fig. 2: Simulation of a parking scenario. The car drives from *Start* to *Goal* following the curve. The numbers indicate at which time it is at which position.

Fig. 3: First time interval of the simulation sequence depicted in Figure 2. The blue line indicates the simulated trajectory without any errors. The red line indicates the trajectory with a strong drift of the position error.

off by several centimeters compared to the trajectory without errors.

In the next time interval, the position error might stay at its value, decrease back to normal or even increase further. The resulting possible paths are depicted in Figure 4. The Figure also visualizes that even with these few error types the number of possible simulation paths grows exponentially with the simulation time: There are three possible paths after one time interval and nine possible paths after two time intervals. In general, the number of simulation paths is:

$$|Paths| = |error\_patterns|^{path\_length} \quad (1)$$

where $error\_patters$ is the set of error patterns that can be applied to the simulation sequence.

Using conventional simulation methods, each of these paths would need to be simulated and hence, the computation time would grow exponentially, too. In order to reduce the computation time, we propose to:

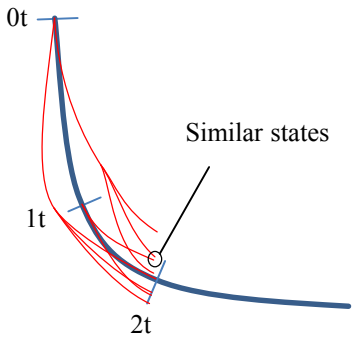


Fig. 4: First two simulation seconds. The red tree indicates the possible trajectories followed for different error patterns. Two of the red lines lead to very similar states. Our concept continues only one of these simulation sequences.

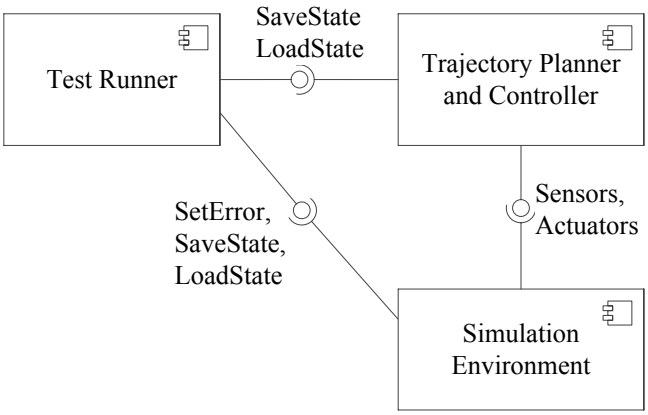


Fig. 5: The architecture of our planning, control and simulation system; The Test Runner component can instruct the planning and simulation components to save or load a state.

- branch the simulation after small time steps using different error types for each branch and
- merge similar states such that only one of them is continued in the simulation.

This way, the number of simulated edges is limited by the resolution of grid used for merging states and the available space, which typically is some region around the trajectory without errors. For example, if the car follows a long street using a stable system that never moves further than 0.1 meters away from the middle of the lane, the simulation will also produce only states in this narrow corridor eventually leading to a number of states growing only linearly with the length of the simulated path.

The approach requires implementing two concepts: Branching the simulation requires being able to save and load the state of both, the simulation environment and all software components involved in trajectory planning and following. Merging simulation states requires determining if two states are similar. Our solution to these two tasks is described in the next section.

As the concept presented in this paper makes no restriction to the simulation environment, it also allows to model almost arbitrary error types. For example, these error characteristics can be applied:

- A delay of the sensor components, for example an obstacle might only be recognized 0.5 seconds after it is in sensor range,
- an actuator delay, for example starting to brake might start only 0.5 seconds after it has been requested,
- errors of the position estimation, like jumps and drifts of the position and orientation error,
- sensor errors, like obstacles being detected at the wrong position, being not detected or detected not existing obstacles,
- actuator errors, like a different acceleration or steering angle being applied,
- or dynamic obstacles occurring in a bad moment.

Each of these error characteristics can be implemented and integrated into the test environment. The integration only requires writing a function that alters the transmitted data accordingly. For the experimental section of this paper we chose to implement jumps of the position errors.

In general, an engineer observing unexpected behavior in one specific situation of the real car can trace it down to sensor and actuator errors, model the behavior for our framework and quickly find out whether this can cause serious problems to any of the tested scenarios. For all error types it can be tested whether the planning and control system works for the worst possible combination of these error types.

In contrast, the straight forward approach not using the concept presented in this paper would be to find the error and adjust the software parameters such that the error does not lead to problems in the currently tested scenario. For example the safety distances could be increased. This activity would not only lose the advantage of using tests performed for one scenario for learning facts about another scenario. Additionally, it might even produce new problems for other scenarios due to the new parameter sets. For example, an increased safety distance might make it impossible to drive through a narrow gate.

IV. Implementation

We implemented the simulation for our trajectory planning and control system. Figure 5 shows an abstract overview over the system. There is a *Trajectory Planner and Controller* component that consists of all components we implemented for the physical car. The *Simulation Environment* includes components representing all sensors and actuators of the car. Finally, the *Test Runner* component manages the execution of the simulation.

In order to do so, the *Test Runner* can command the Trajectory Planner and Controller component to save its currents state and map it to an id specified by the Test

Runner. Saving the State is done using Boost Serialization[1]. This framework can save the current instance and hence the state of any component in the Trajectory Planner and Controller to an archive. Plus, it can restore the state from an archive. After the state is restored, the Trajectory Planner and Controller component behaves as it would have done after saving the state. The same framework is used for the simulation components.

The second part of the Test Runner component is to decide when to save the current state and which state to restore. Figure 6 shows how the Test Runner performs this task. First, it stores the current simulation state (top most box in Figure 6). This is done by sending the save state command to all components and store some meta information about the state inside the Test Runner. Next the current state is compared to all states currently enqueued for further simulation (top most diamond in Figure 6). At the beginning this queue is empty. How the similarity is determined is described in the next paragraph. If there is no similar state and the current state is no terminal state, the current state is enqueued for further simulation. Terminal states are states at the end of a simulation sequence. In the example illustrated for this paper this would be standing at the goal position of the parking lot with a velocity of zero. At this state, our planning and control system sends a status message indicating that the maneuver has been completed. If there is no more state in the simulation queue the whole simulation is finished. If there are states in the simulation queue, the first state of the queue is restored without removing the state from the queue. Restoring the state includes sending the Load command to all components as depicted in Figure 5. Next an error characteristic for the simulation environment is specified. For this task, the Test Runner has a set of error characteristics it applies to all states. Examples of such error characteristics have been listed at the end of the previous section. If the chosen error is the last element of the set, the simulation state is considered fully expanded and hence removed from the simulation state queue. If there are still error characteristics to be applied, the state is left in the simulation queue. With the state restored and the error set, the simulation is run for t_{part} seconds. The cycle time of each component is a multiple of the cycle time of the simulation environment t_{base_cycle}. Each t_{base_cycle} we compare the current system state to the undesired state patterns searched for. In this paper the undesired states are collisions with obstacles. After running t_{part} seconds of the simulation, the system state is stored again as depicted in the top of 6. In our approach we use a constant simulation part size of $t_{part} := 1s$.

As mentioned above, for this algorithm it is necessary to decide, whether two states are similar. For this decision, we use a multi-dimensional grid consisting of the x_1, x_2-position of the car and the orientation. We found these variables to be the most defining properties of a state for the scenarios analyzed for this paper. For other scenarios, this list of variables might need to be extended. For each of these state

[1] http://www.boost.org/libs/serialization

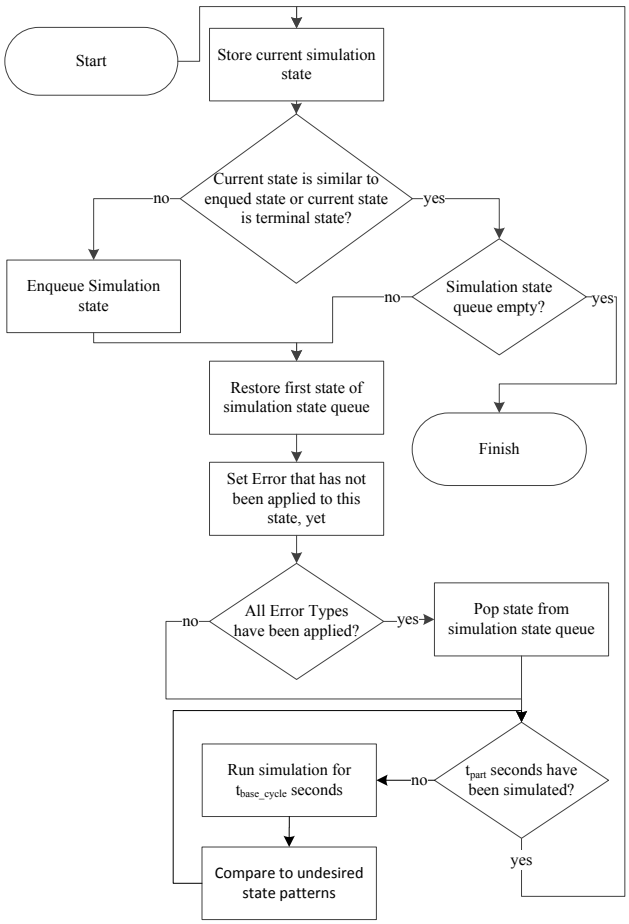


Fig. 6: Algorithm of the Test Runner for deciding when to save or load which state with which error characteristic.

variables a grid size is defined. Two states are considered similar, if they are in the same grid cell. This includes that the difference of each state variable between these two states can be at most the grid size. In our implementation we used the following grid sizes:

$$x_{1,grid} := 0.1m \tag{2}$$

$$x_{2,grid} := 0.1m \tag{3}$$

$$\theta_{grid} := 0.02radians \tag{4}$$

In summary, we have implemented a method for loading and saving states and a managing component comparing states and using the load and save methods to run the simulation.

V. Experimental results

We tested our concept in several scenarios. The first scenario is driving backwards into a narrow passage as depicted in Figure 7. The round shape is the outer shape of the car. The red lines are the walls the car is driving through. They are 3.2 meters away from each other. The thick black line in the middle is a bundle of paths that the car travels in our simulation for different error patterns. The gray box shows it in an enlarged version. The crosses in the middle

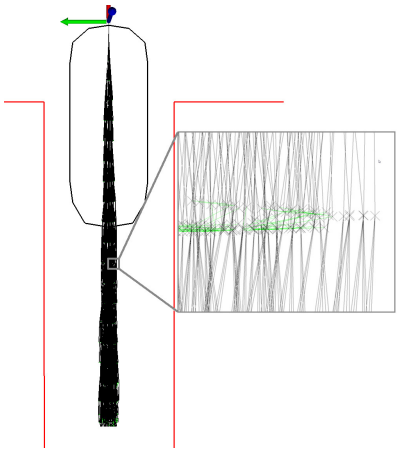
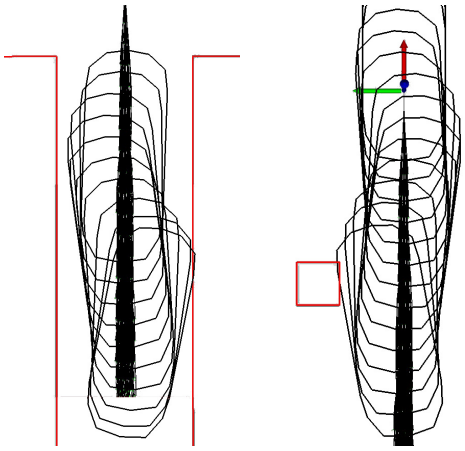


Fig. 7: A narrow passage plotted in ADTF. The big solid line is a tree of possible paths, which the grey box shows in an enlarged version.



(a) wall (b) small obstacle

Fig. 8: Paths covered by the car and leading to a collision with a wall 8a or a small obstacle 8b. They demonstrate the capability of our concept to apply the error patterns at exactly the right time leading to a collision.

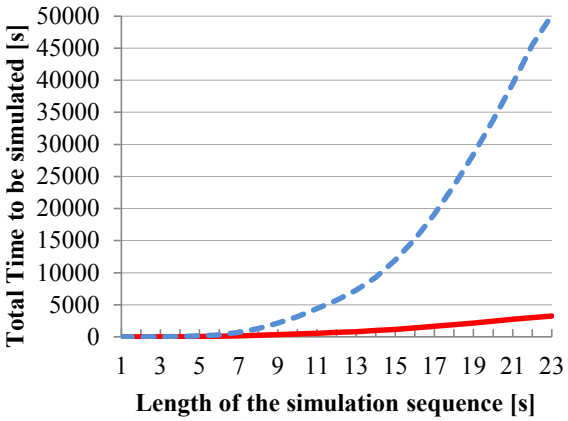


Fig. 9: Total number of seconds simulated using saved states (red solid line) or simulating the whole path every time (blue dotted line). The x axis shows the duration of a single simulation sequence.

are the saved simulation states. The lines are the connections between simulation states. The nodes appear in clusters, as each simulation step lasts one second and the error patterns have no influence on the speed.

In this scenario, the car is ordered to drive through a narrow passage. The planning layer has found a path that can be executed and does not violate the configured safety distances. The pose estimation supplier has specified that the pose estimation may be off by 0.1 meters and might change instantaneously. Hence, the error pattern checked is jumps of the pose estimation by 0.1 meters around the actual position of the car. The simulation has executed 4414 simulation segments, each consisting of 1 second simulated time. As continuous replanning was not used and the system was optimized for a weaker embedded system, the PC (Intel Xeon W3530, 2.8 GHz) was able to execute the segments in 173 seconds. The pattern searched for in the simulation was a collision with one of the walls. The result of the analysis was that the car was colliding while executing state 1572. Figure 8a shows the path leading to this state. At the beginning the pose estimation is off by 0.1 meters to the left. Hence the car steers to the right. At some point the pose estimation error turns to 0.1 meters to the right. Hence the car steers to the left which results in a sufficiently large orientation error to collide with the right wall.

This example also shows that it would not suffice to check with a constant error to the left or to the right, as only the combination of errors to the left and to the right lead to the collision. As it checks every combination of error patterns, the simulation strategy also applies the errors at the right point in time. Figure 8b shows an example in which the error needed to be applied at a very specific point in time. The first four seconds, no error is applied. The following seconds there is first an offset to the left for several seconds and next to the right. If the error would have occurred at any other time, there would not have been a collision.

In these scenarios, the efficiency of the approach presented in this paper can be demonstrated. Figure 9 shows how many seconds need to be simulated for the first n seconds of the scenario. The red solid line shows the approach presented in this paper ascending to the 4414 seconds mentioned above. The blue dotted line shows how many seconds would have been necessary to simulate in an approach using the merging technique described in this paper, but not using saved states. That is, instead of loading a simulation state, the whole simulation until this state would need to be run again. Thus, the number of seconds ascends to more than 50.000. An approach not even merging similar states would require more than one trillion seconds to be simulated which would be impossible.

Additionally, Figure 10 shows the result of a CPU profile created by the open source tool “very sleepy”[2]. It shows

[2] http://www.codersnotes.com/sleepy

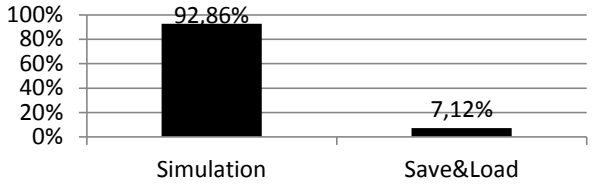


Fig. 10: Time used for the loading and saving and states compared to the time used for the actual simulation

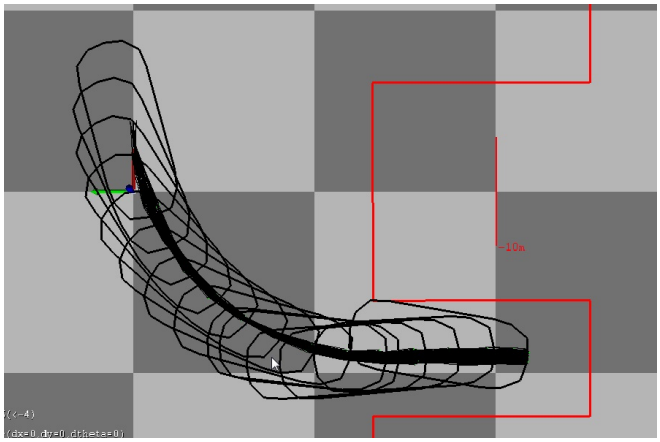


Fig. 11: Parking scenario with a curve analyzed with our concept and similar to the scenario depicted in Figure 2.

that the fraction of time used for loading and saving states is only about seven percent. The remaining time is used for the actual simulation environment and the planning and control system.

As a final example we analyzed a curved parking scenario as depicted in Figure 11 using the same safety distances as in the previous scenarios. The simulation expanded 3608 states in 89 seconds. Figure 11 shows the path that lead to a collision and was detected by our simulation environment.

VI. Conclusion and Outlook

In this paper we presented a new concept for testing a car planning and control system. We demonstrated that this concept is more efficient for finding weaknesses of the planning and control system than sequential simulation. It also scales well with respect to new error patterns, different scenarios and the length of the scenarios. The last point is remarkable, as simulating all error patterns without our concept would result in an exponentially long computation time. For sufficiently long simulation times, the computational effort of our approach grows only linearly with the simulation time.

The concept presented in this paper is the basis for a large set of possible further research. In this paper we present the pioneer work for this concept using a path based controller and a fixed path planning component. The next step would be to cover a trajectory controller and a continuously replanning layer. Together with considering replanning, changes to the starting position of the autonomous vehicle could be addressed as well. Additionally this paper provides the basis for a set of error patterns. We intend to extend this set of error patterns in additional systematic experiments on the real vehicle. Furthermore this paper demonstrates the simulation concept by searching for the event of a collision of the car with an obstacle. Other event patterns are possible like high deceleration or curvature. That is, the comfort of the planning and control system could be evaluated using our simulation concept. For example it could be evaluated that given small errors, the deceleration should never be high.

All in all we presented a new concept for evaluating planning and control systems for advanced driver assistant systems, demonstrated that it performs well and can be the basis for further research.

References

[1] T. Berlin, "Spirit of berlin: An autonomous car for the DARPA urban challenge hardware and software architecture," *Retrieved*, vol. 12, no. 02, p. 2010, 2007.

[2] C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, M. N. Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer, M. Gittleman, S. Harbaugh, M. Hebert, T. M. Howard, S. Kolski, A. Kelly, M. Likhachev, M. McNaughton, N. Miller, K. Peterson, B. Pilnick, R. Rajkumar, P. Rybski, B. Salesky, Y.-W. Seo, S. Singh, J. Snider, A. Stentz, W. R. Whittaker, Z. Wolkowicki, J. Ziglar, H. Bae, T. Brown, D. Demitrish, B. Litkouhi, J. Nickolaou, V. Sadekar, W. Zhang, J. Struble, M. Taylor, M. Darms, and D. Ferguson, "Autonomous driving in urban environments: Boss and the urban challenge," *Journal of Field Robotics*, vol. 25, no. 8, pp. 425–466, 2008.

[3] A. Bacha, C. Bauman, R. Faruque, M. Fleming, C. Terwelp, C. Reinholtz, D. Hong, A. Wicks, T. Alberi, D. Anderson, S. Cacciola, P. Currier, A. Dalton, J. Farmer, J. Hurdus, S. Kimmel, P. King, A. Taylor, D. V. Covern, and M. Webster, "Odin: Team VictorTango's entry in the DARPA urban challenge," *Journal of Field Robotics*, vol. 25, no. 8, pp. 467–492, 2008.

[4] I. Miller, M. Campbell, D. Huttenlocher, F.-R. Kline, A. Nathan, S. Lupashin, J. Catlin, B. Schimpf, P. Moran, N. Zych, E. Garcia, M. Kurdziel, and H. Fujishima, "Team cornell's skynet: Robust perception and planning in an urban environment," *Journal of Field Robotics*, vol. 25, no. 8, pp. 493–527, 2008.

[5] B. J. Patz, Y. Papelis, R. Pillat, G. Stein, and D. Harper, "A practical approach to robotic design for the DARPA urban challenge," *Journal of Field Robotics*, vol. 25, no. 8, pp. 528–566, 2008.

[6] O. Gietelink, J. Ploeg, B. De Schutter, and M. Verhaegen, "Development of advanced driver assistance systems with vehicle hardware-in-the-loop simulations," *Vehicle System Dynamics*, vol. 44, no. 7, pp. 569–590, 2006.

[7] O. Buhler and J. Wegener, "Automatic testing of an autonomous parking system using evolutionary computation," *SOCIETY OF AUTOMOTIVE ENGINEERS INC.*, pp. 115–122, 2004.

[8] J. Wegener and O. Bühler, "Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system," in *Genetic and Evolutionary Computation–GECCO 2004*. Springer, 2004, pp. 1400–1412.

[9] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[10] S. H. A. Niaki and I. Sander, "An automated parallel simulation flow for heterogeneous embedded systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '13. San Jose, CA, USA: EDA Consortium, 2013, pp. 27–30.

[11] A. Aminifar, P. Eles, Z. Peng, and A. Cervin, "Control-quality driven design of cyber-physical systems with robustness guarantees," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '13. San Jose, CA, USA: EDA Consortium, 2013, pp. 1093–1098.

[12] J. Kloos and R. Majumdar, "Supervisor synthesis for controller upgrades," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '13. San Jose, CA, USA: EDA Consortium, 2013, pp. 1105–1110.

[13] Althoff, Matthias, "Reachability analyses and its application to the safety assessment of autonomous cars," Ph.D. dissertation, Technische Universitaet Muenchen, Munich, Feb. 2010.