Available online at www.sciencedirect.com

ScienceDirect

IFAC Papers Online CONFERENCE PAPER ARCHIVE

IFAC-PapersOnLine 49-15 (2016) 044–049

Debugging Autonomous Driving Systems Using Serialized Software Components

Pascal Minnerup * **David Lenz** * **Tobias Kessler** * **Alois Knoll** **

* *fortiss, An-Institut der Technischen Universität München, Germany*
** *Robotics and Embedded Systems, Technische Universität München, Germany*

Abstract: In the development of software-intensive systems in a vehicle, like an autonomous driving system, defects are often only recognized during trials on the physical vehicle. In contrast to a simulation environment, a physically executed maneuver does not offer the possibility to pause and debug critical code sections or to reproduce and repeat faulty trials. Furthermore, development space and capacities are limited inside the car. Therefore, it is best practice to analyze faults observed during a physical execution offline and to reproduce faulty trials in a simulation environment. The repetition in a simulation environment is a time consuming effort but necessary for pushing the software component towards a state in which it showed the faulty behavior. This paper shows an approach for executing the faulty state again in a simulation environment by serializing the exact state of the software system and summarizes practical experience gained by this approach.

Keywords: Fault Detection, Diagnosis, Tolerance and Removal; Path Planning; Advanced Driver Assistance Systems

1. INTRODUCTION

In the last couple of years, the development of advanced driver assistant systems up to highly-automated driving gains more and more weight in the automotive industry. This trend should decrease the traffic injuries and fatalities and offer comfort to the driver and passengers of a car. However, the complexity of automated driving necessitates increasingly large software systems to handle all possible situations. The growing size causes more possible errors that can occur during the runtime and thus increases the time invested for testing and debugging.

When developing such kind of systems, it is common practice to first implement and test the desired behavior of a software component within a simulation environment. First, each software component is considered separately and then the interaction between all necessary modules is tested. This approach allows to pause the simulation at any time a defect occurs in order to attach a debugger and have a look at the internal state of a piece of software.

Unfortunately, when deploying the system to a real vehicle, this approach is not possible anymore. This is especially true for dynamic test cases, where the situation cannot just be stopped or has to be repeated multiple times to find the error. Furthermore, most of the time, the testing of the complete system might be done by developers without the knowledge of how to debug all the programs. Thus, it is necessary to gather data in order to reproduce the failure within a controlled environment by an expert, but the question remains how much data is needed?

There are many influences depicted in Fig. 1 that might cause the occurrence of a defect besides only the input to

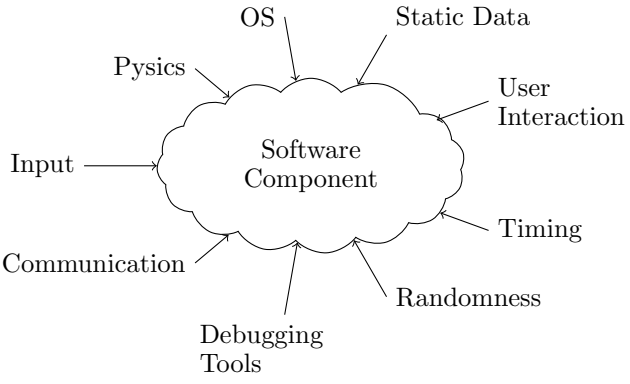

Fig. 1. Different influences on the execution of a software component.

a software component. In a non-realtime runtime environment (which is usually used for predevelopment), timing is a big part that might alter the internal states. For example, results may depend on the order and time of incoming data. This order and time is further influenced by some operating system due to threading, file access, etc. Thus a method is needed in order to reproduce the conditions when the defect occurs as closely as possible and eliminate the influence of the hardware and operating system of the computer in the car.

We propose a framework for serializing internal states of software components to achieve a decoupling of most of the influences presented which lead to a failure within one cycle of a software component.

The structure of the paper is as follows: First, we show the state of the art approach and present approaches from re-

10.1016/j.ifacol.2016.07.612

cent literature to debug complex software systems in general, or automotive functions in particular. Subsequently, we introduce our method of serializing and deserializing of software components and the offline test tools used to find defects. Last, we report some practical experience gained during an industry project applying this solution.

2. RELATED WORK

In industrial projects, failures occurring on physical vehicles are mainly reproduced by recording measured sensor values, actor commands, and debug signals emitted by the software components. In the following, these will be referred to as signal traces. These traces are created with pre development tools like dSpace Control Desk [1], the harddisk recorder in Elektrobit's Automotive Data and Time-Triggered Framework (EB Assist ADTF) [2] or *Bags* of the Robot Operating System (ROS) [3].

Teams researching autonomous driving for the DARPA urban challenge (Defense Advanced Research Projects Agency (2007)) had to cope with failures occurring in the physical vehicle. They dealt with them by recording communication data during the failure event and reproducing the failure by replaying the data. Two of the eleven teams explicitly describe how they reproduced failures (Bacha et al. (2008); Patz et al. (2008)) and seven mention that they are able to record and playback communication data (Chen et al. (2008); McBride et al. (2008); Rauskolb et al. (2008); Bohren et al. (2008); Leonard et al. (2008); Miller et al. (2008); Urmson et al. (2008)) most likely also used for reproducing failures. The remaining two teams do not explain how they dealt with failures (Montemerlo et al. (2008); Kammel et al. (2008)).

In a broader scope of software engineering additional techniques for reproducing failures have been developed. Many approaches are also based on recording and replaying communication data. Clause and Orso (2007) address the problem that communication data logs can be quite large and take as much time to replay as it took to record them. Therefore, they apply techniques for reducing the required size and increasing replay speed. Zamfir et al. (2013) work with still larger amounts of data of data center applications and record only a reduced set of the communication. Plus, they address some sources of non determinism caused by network and scheduler timing. The effect of such non determinism is increased for long replay scenarios. Artzi et al. (2008) eliminate this problem by storing the arguments of all methods called in an annotated java program. This approach works well if the method depends mainly on its arguments. The approach of Rößler (2013) does not depend on recorded data, but tries to reproduce program crashes based on process dumps and randomized test methods. Finally, Yuan et al. (2010) infer information about execution paths for reproducing a failure by matching emitted log messages to lines in the source code. In a following publication (Yuan et al. (2012)), they reduce the number of possible executions by extending the log messages in the source code with additional variables.

[1] http://www.dspace.com
[2] https://www.elektrobit.com
[3] http://www.ros.org/

The main drawback of recording and replaying signal traces is, that the inputs and outputs of a software component only give a hint on the internal states. Many influences mentioned in the introduction in Fig. 1 are not regarded. Thus, due to randomness and other influence factors, the internal states of a replay may drift apart from the previously observed run that lead to a defect.

In contrast, the method presented in this paper:

- allows to exactly reproduce the internal state of a software component at any time
- only needs data of one time instant to reproduce an error as it is not necessary to replay the sequence up to this time instant
- prevents the influence of timing, by decoupling the communication and the cyclic execution
- allows to quickly find executions that clearly lead to an error
- needs less disk space and overhead than capturing signal traces

3. SERIALIZING AND DESERIALIZING SOFTWARE COMPONENTS

The procedure for transferring the state of the software system to an offline simulation environment requires two parts:

- A method to store and restore the state of the software system
- An offline simulation environment to analyze the stored software state.

This section describes the method for storing and restoring the states of the simulation system. First the chosen serialization format is motivated. Section 3.2 describes how the source code is annotated for serialization, followed by a discussion on how the development process is adoped in section 3.3. The final section 3.4 describes when and how the serialization is triggered.

3.1 Choosing the serialization format

Storing the state of a software component means to serialize it. There are several widely used formats for serialized data. For example, Sumaray and Makki (2012) list "XML, JSON, Thrift, and ProtoBuf". For these formats, there are tools for different programming languages to create serialized data. The debugging approach described in this paper has been applied to a component written in C++. Therefore, the selected format has to be supported by tools available in C++. Furthermore, the whole software component is more complex than the data usually transferred over a network connection. Typical tools for creating xml or json files require to explicitly set or read every single data item. Plus, methods for reading and writing potentially hidden information have to be implemented. Google Protocol Buffer [4] additionally requires to write a separate specification of the serialized data. This would be difficult to maintain for a whole software component. In contrast, boost serialization supports complex and nested

[4] https://developers.google.com/protocol-buffers/

data structures. It is designed for persisting data structures and recreating them in "another program context" [5] . In our case, the stored data structure is the entire planning component considered in this paper.

Using boost serialization, any C++ class can be serialized using the command:

Listing 1: Serializing a C++ class

```
1 std::stringstream ss;
2 boost::archive::binary_oarchive oa(ss);
3 oa << planning_component;
```

In this and the following examples, "planning_component" is an instance of the class "PlanningComponent" that is serialized.

In order to make this command work, the serialized software components need to define boost serialization methods. These methods are explained in the next sub section.

3.2 Annotating the source code

Boost serialization offers three options for providing serialization methods for C++ classes:

- Defining serialization methods separately of the class,
- defining serialization methods in the header files, or
- declaring serialization methods in the header files but defining them in a separate file.

For serializing a planning component, the third method is most suitable. The first method requires no changes to the original C++ class. However, it does not allow serialization of private data members and can therefore not be used for all planning classes. The second method results in little implementation effort, but significantly increases compilation time and the length of the header files. This decreases code readability, which should not be necessary for adding debug information. The third method declares serialization methods in the header files, but defines them in a separate file. This way only three lines of code need to be added and compilation increases only negligible. Thefore, we chose it for serializing the planning component.

The three lines of code consist of including a header file, declaring the class to be exported using a boost macro and declaring the serialization method using a custom macro that can be expanded to:

Listing 2: Annotation of the classes to be serialized

```
1 friend class boost::serialization::access;
2 template<class Archive>
3 void serialize(Archive & ar, const uint32_t version);
```

This method can then be implemented in a separate cpp file. Its body is basically a list of member variables of the annotated class.

Finally, there are some components of the software system that should not be serialized because they depend on the current execution environment. For our software, this is the case for the logger and the visualization. Plus, for the clock and the communication middleware some information is serialized, but the deserialization depends on the environment. For example, on the real car, the clock is based on the system clock. During serialization, the current time of the clock is stored. When deserializing it in a simulation environment, the time is restored in a simulated clock. For the communications middleware, unprocessed messages are also serialized.

3.3 Development process

After creating the initial serialization code, it has to be maintained in the development process. If class members of the system's components are added or removed, the serialization method has to be adapted as well. This causes two challenges:

- Compatibility between different versions of the planning system
- Incorrect serialization functions

Boost serialization addresses the first challenge by a version attribute. This attribute allows to serialize an early version of a component and deserialize it to a later version. In particular, this is useful for checking, whether some system state provoking faulty behavior in an early version still produces this behavior in the improved version of a component.

The second challenge to cope with are incorrect serialization functions. Defects in serialization functions can be categorized into two classes:

- Defects leading to crashes of the serialization function
- Defects leading to incomplete serialization

The first class of defects reveals itself and therefore leads only to limited loss of logging data. The second class of defects is more difficult to detect, as the only visible result might be different behavior of the original and the deserialized version of the system component. In order to detect these defects a test suite can compare the behavior of the autonomous driving system with and without serialization. For both classes of defects, the effect is limited, as they are typically unrelated to the phenomenon to be analyzed.

3.4 Triggering Serialization

In our project, three different triggers for serialization can be set up:

- manual trigger for serialization,
- serialization each time, an error message is logged
- serialization before each cycle

The manual trigger can be used to store a state that a test driver considers worth further analysis. For example, without an apparent reason, the car might refuse to continue driving. Instead, if the system recognized that something does not work, the logged error message can be used as a trigger. This is particularly useful to find an example of an already known error pattern. For example situations in which the planning component is unable to find a path. In practice, serialization is performed just before the cycle is executed. If no error occurs in

[5] www.boost.org/libs/serialization

the cycle, the serialized data is discarded. This way the exact situation in which the error occurs can be repeated. Finally, serialization in each cycle means that the serialized data is never discarded. This setting allows to trace back a faulty system state to the point in time in which it became faulty. This setting produces a larger amount of data but still less than a signal trace.

In either of the three triggers for serialization a software engineer notices that some behavior of the vehicle needs further investigation and files a ticket in a bug tracking system. To these tickets they attach the corresponding serialized software components.

4. OFFLINE SIMULATION

The serialized software components can be reconstructed in either a standalone environment for a single software component or in a simulation of the whole planning and control system.

4.1 Trajectory planner debugging environment

We mainly use a standalone environment for reconstructing a serialized software component of the Trajectory Planner. Fig. 2 shows a screenshot of this application. The inputs can be defined using a graphical user interface. In the main area of the window the Trajectory Planner visualizes its results using the following concept. Each software component uses a logging and a visualization interface to emitting information about its computations. This visualization interface is implemented differently depending on the current execution environment. The standalone environment provides an own implementation of the visualization interface. In the standalone environment, the user can accurately control the inputs of the Trajectory Planner including the time of the simulated clock and trigger the execution of another cycle. On the real car the Trajectory Planner is typically executed using a speed optimized release build. The same planner is deserialized in the debugging environment using a debug build. Due to the controlled environment and the debug compilation, the software engineer can step through each line of code of the Trajectory Planner execution using standard debugging software. If the reason for further analysis of this particular state is an error message that has been logged, finding the problem is particularly simple. In this case, the software engineer adds a break point to the corresponding line of code producing the error message and analyzes the state of the component when the debugger hits the break point.

4.2 Full simulation

Some cases require to execute the particular situation in a full analysis. For example, if the deviation between the planned path and the driven path is unusual high it can be relevant to analyze whether known inaccuracies of sensors and actuators that are modeled in the simulation environment can explain these deviations. In this case, instead of a single component, the whole planning system is deserialized in the simulation environment. Physical systems like the vehicle itself and black-box software components have to be replaced by sufficiently accurate models. Minnerup et al. (2015) describe this process based on the methods published by Minnerup and Knoll (2014).

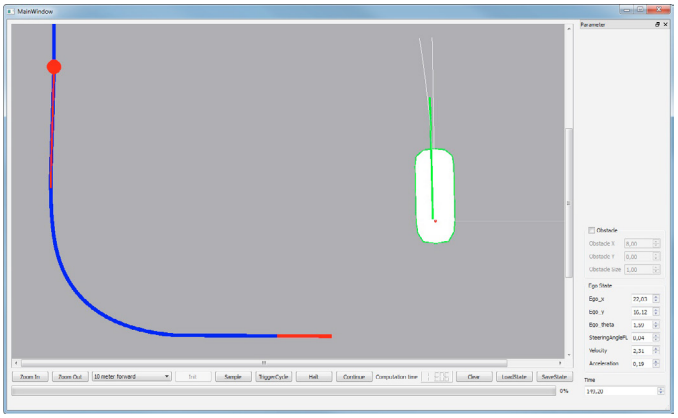

Fig. 2. Deserialized planner in a standalone environment. The main area visualizes the planned path and the current position known by the planner.

5. PRACTICAL EXPERIENCE

We apply the debugging concept presented in this paper in a project with Audi. In this project, we develop a planning and control system that is integrated in several sub projects, for example an autonomous parking garage (Lenz et al. (2014)). For each of these projects there is an integration team that executes the planning and control system developed by our team together with components from other teams. If the integration team encounters some undesired behavior, it files a ticket in a bug tracking tool and attaches the log messages produced by the planning and control system. Several problems were solved by re-executing the problematic situation based on a serialized planning component. Section 5.2 describes one such problem. Section 5.1 shows that the size of serialized software components is smaller than the size of ADTF signal traces. For some tickets the serialized planning components were also useful for understanding that the planning component worked correctly.

Additionally to the bug reports we perform own tests on the physical vehicle. For each test day, we store the serialized planning components and analyze unusual behavior. This unusual behavior includes not only faulty behavior, but also patterns that we did not expect and need to understand. This leads to finding defects that affected the performance of the planning components but did not lead to obviously wrong behavior.

5.1 Simulation Results

The first test performed for validating the concept presented in this paper is to run it in a simulation environment. During the simulation, the Trajectory Planner is serialized each cycle and all inputs of the Trajectory Planner are stored as a signal trace using the ADTF HarddiskRecorder. In a seven minute simulation, the Trajectory Planner executed multiple parking maneuvers. For this time, the ADTF signal trace required 812 MB of storage, while all serialized Trajectory Planning components required only 10 MB. The maximal size of a single Trajectory Planning instance was 0.016 MB. Hence, this experiment demonstrates that the serialization concept requires less storage than a signal trace, although it provides more

information for debugging purposes as shown in the next sub section. The computation time necessary for serializing a local planner was always less than 1 ms.

5.2 Case Study

Figures 2-6 illustrate the process of tracing back a software failure to a fault and fixing this fault. During an integration test, the integration team notices that the car drives less smooth than usually. Therefore they send the corresponding log files to the team developing the planning and control component for further analysis. We start by identifying the uncomfortable driving situation in the recorded acceleration data included in the logs and shown in Fig. 3. It depicts the measured acceleration and the acceleration computed by the planning component. Based on interior system knowledge, we know that at the marked point, this planned and measured acceleration should almost meet, which they do not. This difference indicates that the planning component emits a wrong acceleration.

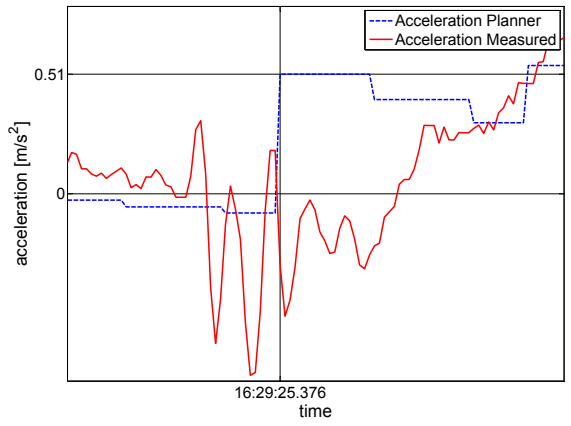

Fig. 3. Planner assumed the wrong current acceleration: At the marked point in time the planned and the measured acceleration should be identical.

Analyzing the log messages written by the planning component at the detected point in time indicates that the planning component assumed a wrong initial acceleration as Fig. 4 shows. The difference to the value highlighted in Fig. 3 is due to an interpolation.

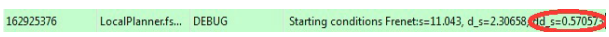

Fig. 4. The log message of the planning component confirm the wrong acceleration assumption (red circle).

From the serialized planning components we now load the one that was serialized just before the cycle emitting the suspicious log message was executed. This file is deserialized in the standalone environment shown in Fig. 2. It visualizes the current state of the planning component including its current position, the current reference path and the plan for the next seconds. In this case the visualization does not indicate any problem.

Stepping through the execution of the planning cycle with a debugging tool attached finally leads to the code fragment shown in Fig. 5. The shown values of the local variables indicate that an interpolation function does not return the expected result. The expected interpolated value is in between the two input values.

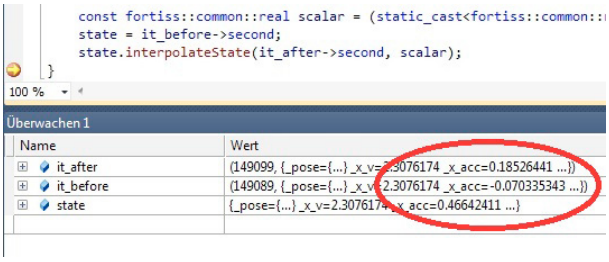

Fig. 5. The standalone environment allows to re-run the cycle using a debug build of the planner based on the serialized state. Using a standard debugging tool allows to quickly find the line of code and variable values leading to a bad acceleration interpolation.

After correcting the corresponding code fragment, the same serialized state can be executed with the corrected planning component. At the regarded position, the debugging tool shows that the wrong behavior does not occur anymore.

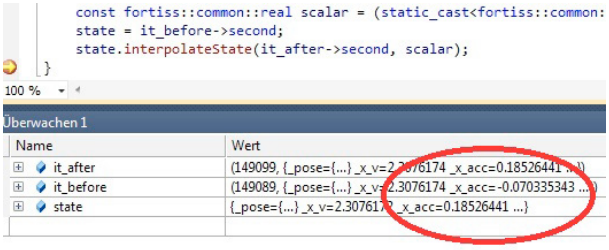

Fig. 6. After fixing the corresponding function, the serialized situation can be repeated with a different version of the planner showing the problem does not occur anymore.

As this example shows, we have been able to trace back an inaccurate error description (the vehicle acts uncomfortable) to a specific defect (incorrect implementation of an interpolation function) in the planner code. An additional unit test avoids further problems in this piece of code. Here a signal trace would have been less useful:

- It would require significantly more time until the planning component would have reached the exact same state as in the vehicle,
- the error was strongly related to the timing of the components: A few milliseconds difference let the problem disappear,
- A signal trace would have been very large and might not have been transferred on the internet

5.3 Discussion

The concept presented in this paper performs best for components with significant state information. In this case it is more useful than a signal trace. The storage advantages compared to a signal trace are maximized if the cycle frequency is low compared to the frequency of the input data. Signal traces are still useful, if the change of

a variable needs to be visualized, which is often necessary for controller development.

6. CONCLUSION

In this paper, we have demonstrated a method for debugging failures occurring during a test drive with a physical vehicle in an offline test environment. Using the serialization features of the boost C++ library we are able to completely save and reload the system state, to reproduce the observed error afterwards, and find a defect in the source code.

The demonstrated strategies have been developed and are used by our development team in an industry scale project implementing several autonomous driving scenarios on multiple test vehicles. The initial and continuing overhead maintaining the serialization code for the whole planning system has proven to be beneficial as tests drives can mainly be used for data acquisition and parameter optimization. Bug fixing can be shifted to the office.

Furthermore the development cycle is accelerated as the maximum amount of information during a test drive is persisted for further analysis by an expert and additional test drives are avoided. Even timing aspects and internal system states can be analyzed afterwards. Note that, as also seen in the case study 5.2, logging error messages and recording characteristic signals can be beneficial.

Currently, the serialization concept is used for the planning components of an autonomous vehicle running on PC-like hardware. We aim to implement a similar concept for the control system executed on rapid prototyping real-time hardware which brings along tighter restrictions.

REFERENCES

Artzi, S., Kim, S., and Ernst, M.D. (2008). *ReCrash:Making Software Failures Reproducible by Preserving Object States Shay*, volume 5142 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg.

Bacha, A., Bauman, C., Faruque, R., et al. (2008). Odin: Team VictorTango's entry in the DARPA Urban Challenge. *Journal of Field Robotics*, 25(8), 467–492.

Bohren, J., Foote, T., Keller, J., et al. (2008). Little Ben: The Ben Franklin Racing Team's entry in the 2007 DARPA Urban Challenge. *Journal of Field Robotics*, 25(9), 598–614.

Chen, Y.L., Sundareswaran, V., Anderson, C., et al. (2008). TerraMax: Team Oshkosh urban robot. *Journal of Field Robotics*, 25(10), 841–860.

Clause, J. and Orso, A. (2007). A Technique for Enabling and Supporting Debugging of Field Failures. In *29th International Conference on Software Engineering (ICSE'07)*, 261–270. IEEE.

Defense Advanced Research Projects Agency (2007). Urban Challenge - Rules. Technical report.

Kammel, S., Ziegler, J., Pitzer, B., et al. (2008). Team AnnieWAY's autonomous system for the 2007 DARPA Urban Challenge. *Journal of Field Robotics*, 25(9), 615–639.

Lenz, D., Minnerup, P., Chen, C., and Roth, E. (2014). Mehrstufiges Planungskonzept fuer pilotierte Parkhausfunktionen. In *30. VDI/VW-Gemeinschaftstagung "Fahrerassistenz und Integrierte Sicherheit 2014"*. Wolfsburg, Germany.

Leonard, J., How, J., and Teller, S. (2008). A perception-driven autonomous urban vehicle. *Journal of Field Robotics*.

McBride, J.R., Ivan, J.C., Rhode, D.S., et al. (2008). A perspective on emerging automotive safety applications, derived from lessons learned through participation in the DARPA Grand Challenges. *Journal of Field Robotics*, 25(10), 808–840.

Miller, I., Campbell, M., Huttenlocher, D., et al. (2008). Team Cornell's Skynet: Robust perception and planning in an urban environment. *Journal of Field Robotics*, 25(8), 493–527.

Minnerup, P., Kessler, T., and Knoll, A. (2015). Collecting Simulation Scenarios by Analyzing Physical Test Drives. In *18th IEEE International Conference on Intelligent Transportation Systems*, 2915–2920.

Minnerup, P. and Knoll, A. (2014). Testing autonomous driving systems against sensor and actuator error combinations. In *Intelligent Vehicles Symposium Proceedings*.

Montemerlo, M., Becker, J., and Bhat, S. (2008). Junior: the stanford entry in the urban challenge. *Journal of Field Robotics*.

Patz, B.J., Papelis, Y., Pillat, R., Stein, G., and Harper, D. (2008). A practical approach to robotic design for the DARPA Urban Challenge. *Journal of Field Robotics*, 25(8), 528–566.

Rauskolb, F.W., Berger, K., Lipski, C., et al. (2008). Caroline: An autonomously driving vehicle for urban environments. *Journal of Field Robotics*, 25(9), 674–724.

Rößler, J. (2013). *From software failure to explanation*. Ph.D. thesis, Universität des Saarlandes.

Sumaray, A. and Makki, S.K. (2012). A comparison of data serialization formats for optimal efficiency on a mobile platform. In *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication - ICUIMC '12*, 1. ACM Press, New York, New York, USA.

Urmson, C., Anhalt, J., Bagnell, D., et al. (2008). Autonomous driving in urban environments: Boss and the Urban Challenge. *Journal of Field Robotics*, 25(8), 425–466.

Yuan, D., Mai, H., Xiong, W., et al. (2010). SherLog: Error Diagnosis by Connecting Clues from Run-time Logs. *ACM SIGPLAN Notices*, 45(3), 143.

Yuan, D., Zheng, J., Park, S., Zhou, Y., and Savage, S. (2012). Improving Software Diagnosability via Log Enhancement. *ACM Transactions on Computer Systems*, 30(1), 1–28.

Zamfir, C., Altekar, G., and Stoica, I. (2013). Automating the debugging of datacenter applications with ADDA. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 1–12. IEEE.