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

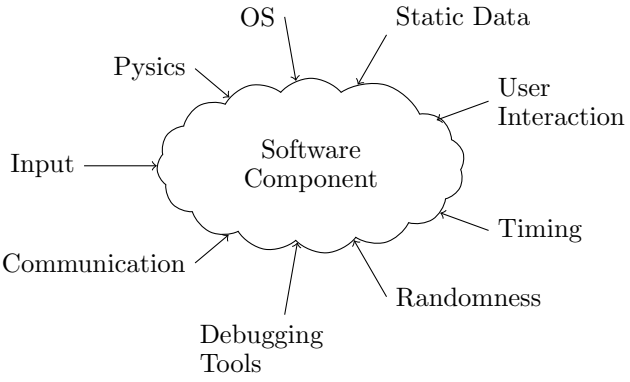

Fig. 1. Different influences on the execution of a software component.

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/