

Efficient Safety Analysis of Automotive Software Systems

Mario Trapp, Soeren Kemmann and Ralf Kalmar
Fraunhofer Institute for Experimental Software Engineering, Germany

Christian Denger
Siemens AG, Germany

Copyright © 2009 SAE International

ABSTRACT

Software has rapidly gained importance as a driver for innovation in automobiles. Since many safety-related automotive systems make intensive use of software, the upcoming ISO 26262 poses several requirements addressing software development, including safety analyses for software. For software, however, safety analysis techniques are seldom applied in practice. It is unclear how to apply them and in many cases even their usefulness in general is questioned.

This article illustrates why software safety analyses are indispensable, how they can be efficiently applied to complex systems, and how they relate to existing software quality assurance techniques and system safety analyses.

INTRODUCTION

The automotive safety standard ISO 26262 [1] has evolved for several years now and will probably be published as draft international standard (DIS) in 2009. ISO 26262 is a domain-specific adaptation of the IEC 61508 and will affect the development of all road vehicles. ISO 26262 focuses on electric and electronic systems and particularly includes requirements for software development. In addition to requirements affecting traditional development and quality assurance techniques - like the demand for coding guidelines, or

the application of special test techniques - the standard explicitly requests qualitative safety analyses for software architecture as well as unit design. This requirement has reignited passionate discussions about the sense, purpose, and feasibility of safety analyses for software. Leaving out the problem of quantitative analyses of software systems, which are not demanded by standards, most discussions about software analyses end up in two major arguments: (1) It is often unclear what additional benefit a software safety analysis yields to existing safety analyses at the system level on the one hand and to existing software quality assurance processes on the other hand (2) It is unclear how safety analyses can be economically applied to software: It is argued that software is much too complex, resulting in too much effort and cost. Sometimes both arguments are combined arriving at the conclusion that software safety analyses cause enormous additional effort without producing any additional benefit.

In order to provide answers to these questions and to provide a sound basis for further discussions, this article gives a systematic overview of software safety analyses, why they are very important, and how they can be applied efficiently. We point out typical challenges encountered by developers during the introduction of software safety analyses and how they can be solved. To this end we exemplarily sketch two approaches that have been successfully applied in different industries in order to overcome typical challenges of software safety analyses.

We first illustrate our understanding of the role of software in the development of safety-related systems. This includes the identification of issues that must be addressed in order to ensure the safety of a software-intensive system. Moreover, we discuss which of these issues must be addressed by software safety analyses. As a further aspect, the second chapter provides an overview of the state of the art of software safety analyses and a discussion with respect to other quality assurance and safety activities. Chapter “SOFTWARE SAFETY ANALYSES IN PRACTICE” illustrates how software safety analyses can be performed efficiently.

SOFTWARE SAFETY

ISO 26262 defines safety as the “absence of unacceptable risks” [1], which is very similar to the definition “freedom from unacceptable risk” as found in IEC 61508 [2]. In both cases, safety is defined based on the term risk, which is defined as a “combination of the probability of occurrence of harm and the severity of that harm”. A hazardous event leading to harm occurs if the system fails with certain failure modes in specific situations. Obviously the hazardous events relate to system failure modes like undesired braking of a single wheel while driving. For these reasons, safety is a system-level property. Nonetheless, major functionalities of a vehicle are realized in software, so that software faults are causes of hazardous events at the system level. Consequently, software safety can be defined as the “freedom from faults of software used in a specific operational environment that have the potential to cause hazardous events” [3, 4]. Based on these definitions, one major objective of software safety engineering is the reduction of the residual risk resulting from software faults to an acceptable level.

It is further important to understand the relationship between software faults and hazardous events at the system level that might actually lead to a hazard (i.e., damage). To this end, Figure 1 presents a so-called failure cascade, which is described in similar variants by various authors (e.g., [3, 5, 6, 7, 8]).

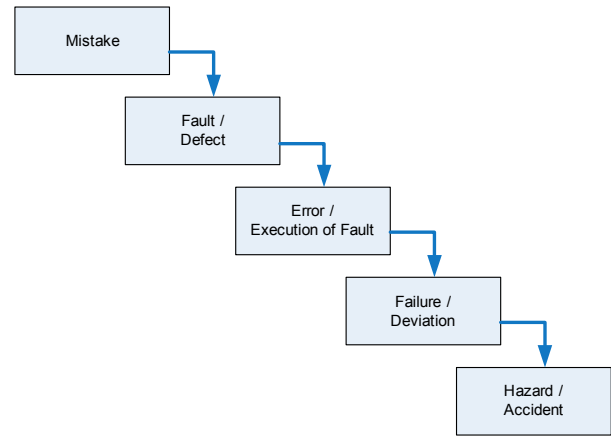


Figure 1: Relationship between software fault and system hazards.

The cause of any software fault is a (human) *mistake*, which can occur in any development step. An example is misinterpretation of the needs of a customer by a requirements engineer. This mistake becomes a *fault* as soon as it manifests in a software work product. For example, the misinterpretation of the requirements leads to an incorrect algorithm specification. It is important to note that a fault can be included in all software work-products, such as requirements, interface specification, architectural models, or software code. In case the fault in the software code and is executed, the fault becomes an *error*. An error is defined as a deviation from a desired or intended system state [3]. Hence, when a software fault is executed the system including the software is put into an erroneous state. This error might lead to *failure* if it is not detected and handled at runtime. A failure is defined as a deviation from the specified behavior. Since most systems are defined hierarchically, a failure in one component might be the fault in a super-ordinate component leading to a cascade over the different hierarchy levels. Only if a software failure propagates to the system border, leading to an undesired or unexpected system behavior, the software fault leads to a system failure. On the system level, the deviation from the expected behavior can then cause a hazardous event, and in the final consequence, an accident leading to harm.

Having this cascade in mind, the risk resulting from software faults can be reduced by either preventing or repairing software faults before the software is released (fault avoidance) and by appropriate means for detecting and handling errors in order to contain errors prior to their propagation to the system border (failure avoidance).

In practice, software faults are often considered to be implementation, respectively coding, faults only. It is assumed that a good quality assurance process will eliminate these faults to a sufficient extent so that the failure propagation can be contained from the very

source. Based on this understanding of software faults, some people might think that software quality assurance was sufficient to achieve the above-mentioned purpose of software safety engineering. This is one of the main reasons why the necessity of additional software safety analyses is questioned in general. Indeed, software quality assurance is of crucial importance and consequently also a major topic addressed in ISO 26262. However, the causes of software failures are much more manifold than just being implementation faults.

The different possible causes of software errors are shown in Figure 2. First it must be considered that software errors are not only caused by software faults but also by unavoidable external faults, particularly hardware failures like bit-flips in a memory caused by EMR. Therefore, software errors are unavoidable and it is of crucial importance to understand how failures propagate in the system in order to choose sufficient and required countermeasures.

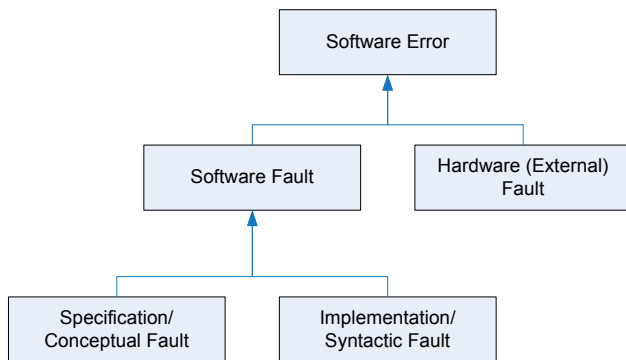


Figure 2 Sources of software errors

Second, one must differentiate between implementation/syntactic faults and specification/conceptual faults. *Implementation faults* mean that the correct specification has been realized wrongly. This includes not only source code, but all work products created in the software development lifecycle. Causes for implementation faults could be a mistake made by the developer or a fault in a tool like a code generator. If, for example, a variable has not been correctly initialized, this might lead to errors and consequently to failures. *Conceptual faults* mean that the specification is incorrect. Even if implemented correctly, there is nonetheless a conceptual fault in the system. If we take a very simple vehicle speed computation based on wheel speeds as an example, the speed will be erroneous as soon as there is wheel slip, even though the algorithm has been implemented correctly. These aspects are studied in more detail in the following.

FAILURE PROPAGATION – Software errors are unavoidable. Even if we assume perfect quality assurance, software errors may have various external causes, like hardware or sensor faults. External faults usually cannot be controlled and are not covered by software quality assurance activities like testing or inspections. Instead, it is necessary to have special runtime error detection and error handling mechanisms as it is also demanded by most safety standards, including ISO 26262. Moreover, it is often not possible to avoid conceptual faults for technical and/or economic reasons. Understanding how such faults will cause a software error and how, in turn, this error might result in a system failure is therefore indispensable for understanding the impact on system safety and thus for choosing appropriate countermeasures. A good understanding of the cause-effect relationships of failure propagation in a software system is essential for systematically assessing if all possible faults are covered by appropriate measures. In the sense of efficient safety engineering, this understanding is required in order to arrive at a systematic and verifiable decision, regarding which of the currently applied measures are actually required to ensure system safety. Since the failure propagation in a software system easily becomes very complex, it is neither possible to ensure its safety nor to choose reasonable countermeasures without a systematic analysis – a software safety analysis. This means that one of the main purposes of software safety analysis is the understanding of the cause-effect relationships of failure propagation in the software system. In this context it is necessary to understand how software faults and external faults lead to failures at the system level in order to identify potential hazards that have not been considered (completely or accurately). On the other hand it is often useful to identify potential causes of system-level hazards in the software in order to apply reasonable countermeasures, i.e., a set of actually required and sufficient failure prevention actions.

As an example, let us look at an emergency stop function of a remote controlled safety concept car¹. The purpose of the function is to brake the car to a stop as soon as the signal of the remote control is lost. As a possible hazard, the function could cause brake forces that are too high, leading to instability of the car.

¹ In order to comply with non-disclosure agreements, we have transferred examples found in real systems to fictive examples based on a RC-scale concept car (cf. Figure 3).

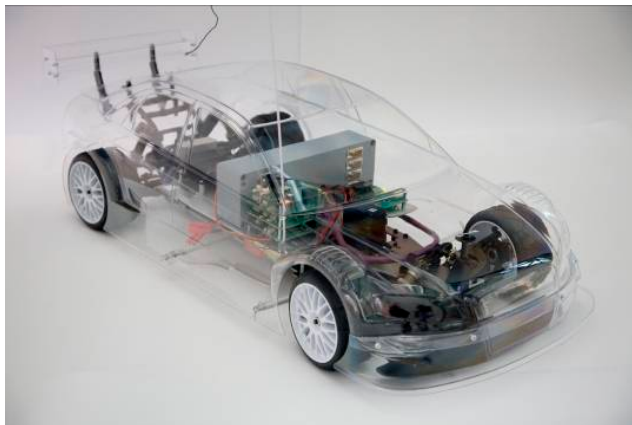


Figure 3 RC-Concept Car used as example

During a software safety analysis, the failure propagation is analyzed. It turns out that a failure of the components might lead to an undesired acceleration of the car because the respective software component decelerates the car by accessing a generic longitudinal acceleration interface. An initial system-level hazard and risk analysis however, had not identified this particular incident, since it was – implicitly or explicitly – assumed that the function would only brake the car.

We found comparable issues in several different real-world systems, including functionalities with the highest safety integrity level assigned (ASIL D), and it always turned out that the error detections were not sufficient, since they were tailored to the known failure modes and neglected the additionally found mode. The safety analyses at the system level had not revealed this issue, since they regarded software as a black box. The fault, however, was inside the software. The quality assurance had not revealed this problem, since it focused on verifying the functional specification. Mapped to our fictive example, this means that the case of an undesired acceleration had not been specified as a potential failure mode and had thus not been subject to verification activities. Considering the complexity of today's automotive software systems and the intensive interaction between various different software components, finding such issues by chance during a review is nearly impossible.

CONCEPTUAL VS. REALIZATION FAULTS – As mentioned above, it is further important to differentiate between specification/conceptual faults and implementation/syntactical faults (cf. Figure 2). Typical examples of implementation faults are uninitialized variables, an endless loop in the source code or an ambiguous transition guard in a state chart, meaning that developers made a mistake while implementing a specified functionality. Many quality assurance techniques are focused on the identification and elimination of implementation faults. Consequently

additional safety analyses also focusing on implementation faults do not provide any added value.

Many severe accidents in recent years however were not caused by implementation faults but by conceptual specification faults [3], [9], [10]. This means that developers did not make an implementation mistake during development, but they implemented a functionality that was unsafe by definition. If we again regard our fictive example, many functions make use of a single-track model. Although this model is correctly implemented, the car shows an undesired behavior in certain situations. Obviously, such a model approximates the physical properties of the real world and is only valid if several different preconditions are fulfilled. For example, calculations based on a single-track model are only correct as long as the lateral acceleration of the car remains below a given threshold, which is not the case in many driving situations. This means that there is an explicit fault in the system leading to an error in certain driving situations. As part of the safety analysis it must be analyzed in which situations these preconditions are not fulfilled and how likely these driving situations are. By analyzing the failure propagation as described above, it is possible to identify consecutive errors and eventually assess the effects at the system level.

It is often argued that this kind of analysis is not a software safety analysis but a functional safety analysis. As mentioned earlier, however, software is more than source code and software safety analysis must cover all artifacts created during the software development process. This is particularly true since often the only given software specification is the software requirements specification. Based on the latter, the software system is refined into single functions, which are decomposed into single components of a functional chain from sensors to actuators. These components are, in turn, further decomposed hierarchically. This means that the system is refined incrementally and at each refinement step, a decision is made on how the given task of a component can be realized. In this process, it is very likely that potential conceptual faults like approximations, assumptions on driving situations, usage conditions, or the quality and semantics of used signals are inserted. In many cases found in practice, the decisions made during these refinement steps are neither documented nor is their impact on safety analyzed systematically. Although the engineers certainly took care of the impact, it was usually their main intention to implement a working system and not to get a safe system. Only in a few cases the analysis is done systematically or even documented. At the latest when the system is extended or modified and the author of a model or source code is not available anymore, assumptions and approximations made are often lost, leading to an incalculable risk. It is therefore necessary to systematically analyze different runtime conditions and to try to document the most