Evaluation of Fault Tolerance Mechanisms with Model Checking

Vince Molnár, István Majzik
Budapest University of Technology and Economics
Department of Measurement and Information Systems
Budapest, Hungary
Email: {molnarv, majzik}@mit.bme.hu

***Abstract*—Failure Mode and Effects Analysis (FMEA) is a systematic technique for failure analysis. It aims to explore the possible failure modes of individual components or subsystems and determine their potential effects at the system level. Applications of FMEA are common in case of hardware and communication failures, but analyzing software failures (SW-FMEA) poses a number of challenges. Failures may originate in permanent software faults commonly called bugs, and their effects can be very subtle and hard to predict, due to the complex nature of programs. Therefore, an automatic method to analyze the potential effects of different types of bugs is desirable. Such a method could be used to automatically build an FMEA report, or to evaluate different failure mitigation techniques based on their effects on the outcome of faults. This paper follows the latter direction, demonstrating the use of a model checking-based automated SW-FMEA approach to assess error detection mechanisms of safety-critical embedded operating systems.**

***Index Terms*—Failure Mode and Effects Analysis, SW-FMEA, model checking, fault tolerance, error detector**

I. Introduction

Safety and in particular the risk of failure is one of the main concerns of safety-critical systems. Certification requires the systematic analysis of potential failures, their causes and effects, and the assessment of risk mitigation techniques used to reduce the chance and the severity of system-level failures.

One of the first systematic techniques for failure analysis was Failure Mode and Effect Analysis (FMEA) [2]. FMEA is often the first step in reliability analysis, as it collects the potential failure modes of subsystems, their causes and their effects on the whole system. Together with criticality analysis (often treated as part of FMEA, sometimes emphasized by the term FMECA), the output of FMEA serves as the basis of other qualitative and quantitative analyses, as well as design decisions regarding risk mitigation techniques.

FMEA is usually applied at the hardware and communication level, where it requires a qualified analyst to collect postulated component failures and identify their effects on other components and the system level. In case of software (SW-FMEA), failure modes originate in different types of programming faults, commonly referred to as bugs. Due to the complex nature of software and the many types of potential bugs, it is much harder to collect failure modes and deduce their potential effects, so an automated mechanism is desired.

This paper presents a way of automated SW-FMEA with the use of executable software models. Assuming a set of predefined fault types (programming errors) and a specification of safe behavior at the system level, the proposed approach applies model checking to generate traces leading from fault activations to states that violate the specification (system-level failures). These traces can be used to understand and demonstrate fault propagation through the system and also as test sequences for the final product.

In addition to automated SW-FMEA, the proposed approach can be used to evaluate the efficiency of fault tolerance and error detection mechanisms. Compared to the baseline of having only the core system model, fault tolerance mechanisms should mask as many faults as possible (reducing the number of fault activations that can lead to a system-level failure), while error detectors should catch the propagating error on as many traces as possible. The evaluation of the latter mechanism is presented on a case study, using a model of the OSEK API specification [1], which is a commonly used interface specification for embedded operating systems.

The paper is structured as follows. Section II introduces the key concepts of FMEA and model checking, then a framework for model checking-based FMEA is outlined in Section III. Applications of the approach are discussed in Section IV, while Section V presents a case study. Section VI provides the concluding remarks and our directions for future work.

II. Background

This section summarizes the main idea of FMEA and in particular SW-FMEA, as well as model checking that is the basis of the approach presented in Section III.

A. Failure Mode and Effects Analysis

FMEA involves *1)* the enumeration of potential failure modes of subsystems, *2)* an inductive reasoning of their effects on different levels of the whole system (called *error propagation*), and *3)* often the deductive analysis of their root causes. The analysis is usually based on a model or specification of a component, as well as historical data and experience with similar components. The result is recorded in an FMEA spreadsheet. Failure modes are then categorized based on criticality, representing the level of chance and the severity of potential consequences. Criticality can prioritize failures, and based on the discovered causes and effects, fault-tolerance or

error detection mechanisms can be designed to mask or detect faults to ensure fail-safe operation of the system.

There are three main concepts related to error propagation. A *failure* is an incorrect system function, i.e., an observable invalid state. An *error* is a latent invalid state that has no observable effects yet. Finally, a *fault* is the cause of a failure, which can be either some kind of defect (physical or design) or the failure of a related subsystem.

During error propagation, an *activated fault* can cause an error, which will turn into a failure once it becomes observable (e.g., by crossing an interface). FMEAs are usually performed on many levels during the design of a system, so a failure of a component is often a fault in another one. FMEA usually assumes that only a single failure mode exists at a time.

Software FMEA: When performing FMEA on software components, failure modes are usually caused by programming (or configuration) errors. The challenge of analyzing them is twofold. First, it is very hard to come up with a realistic set of programming faults (called a *fault model*). The source of bugs is almost always a human, and the most typical faults highly depend on the programming language and the domain as well. Constructing a realistic fault model is therefore even harder in case only a design model is available. Secondly, the effects of a bug is hard to track as it evolves in a complex system.

This paper focuses on the challenge of analyzing the effects of programming faults. The problem of designing proper fault models is not discussed here, we refer to the fact that it is also an important challenge in the field of *mutation-based testing*. For an extensive overview of mutation-based testing and fault models, the reader should refer to [7].

Most of the previous approaches to SW-FMEA build on software testing (e.g., [8]), injecting faults directly into the program code and running a set of tests to see any possible global effects. Model-based SW-FMEA has also been proposed recently [4] based on executable software models, model-level fault injection and simulations. Another approach, similar to the one presented in this paper, uses model checking to detect the violation of the system-level specification in case active faults are present [5]. The common features of these approaches include a predefined set of faults injected into the code or model, a description of system-level failures/hazards, and some sort of execution (either testing, simulation or model checking) to generate traces connecting the first two.

Our approach presented in Section III improves existing model checking-based SW-FMEA by optimizing fault activations and using monitors instead of a formal specification.

B. Model Checking

Model checking is an automated formal verification technique used to verify whether a system satisfies a requirement or not. This is done by systematically (and typically exhaustively) analyzing the states and/or possible behaviors of the system model (i.e., the *state space*). If the specification is violated, model checkers prove it with a *counterexample*.

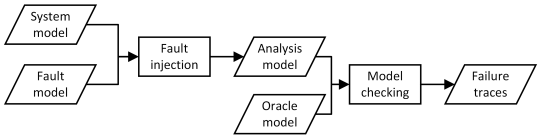

Fig. 1: Overview of the presented approach.

In this work, the tool SPIN was used as a model checker [6]. SPIN is an explicit model checker (using an explicitly stored graph representation of the state space) capable of reachability analysis (is there a reachable "bad" state?) and linear temporal model checking (describing complex temporal behavior). Its strengths include its maturity, the rich set of configuration opportunities and the expressiveness of its input model, given in PROMELA (PROcess MEta LAnguage).

III. Model Checking-based Software FMEA

The approach presented in this paper focuses on the "Effect Analysis" part of FMEA. Assuming a set of possible faults (failure modes) in the software and a characterization of system-level failures, it examines an executable model of the system to generate traces leading to system-level failures.

The process (shown in Figure 1) starts with fault injection, when the input model is transformed into an analysis model containing faults that can be turned on or off. It is assumed that there is an oracle model that allows the detection of system level failures (see Section III-2 for details), so the model checker can analyze the model to check if any fault can cause a system-level hazard. The output is a set of traces that lead from every dangerous fault activation to reachable system-level failures.

1) Fault Injection: The method requires a fault model in terms of the modeling language. Here, a fault model is assumed to be a set of alterations (mutations) that can be applied on the model. The actual alteration is performed by adding a *trigger variable* to activate or deactivate the fault, i.e., with the trigger variable set to *false*, the model should behave correctly, while a value of *true* should cause an erroneous state when the affected part is executed. Note that trigger variables become part of the system as auxiliary state variables.

Using the trigger variables, a number of different fault types can be modeled. First, a fault can be permanent (only nondeterministic $false \rightarrow true$ transitions) or transient (nondeterministic $true \rightarrow false$ transitions are also present). Although in case of software bugs, the faults are usually permanent, it is sometimes useful to have transient faults to simulate the effects of hardware faults as well. Second, it is sometimes desired to restrict the number of faults in the system to at most one, or in some cases at most two.

By injecting the fault activation mechanism into the model, a model checker is free to choose which fault to activate by setting the trigger variables as long as it meets the restrictions.

2) Failure Detection: The traditional approach in model checking is to provide a formal specification of the system. Automated FMEA can then check if the specification still holds in the presence of faults (as described in [5]).

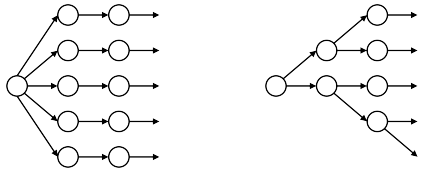

Fig. 2: Shape of the state space with eager and lazy evaluation.

In this work, we suggest an alternative that is closer to a safety engineer's viewpoint. Instead of specifying a failure (or the correct behavior), it is sometimes easier to model a component to detect and signal requirement violations. Such a model can be idealistic (e.g., it may observe every detail of the system or have infinite memory), since its only role is to provide a definition for system-level failures, it does not have to be implemented in the real system. Due to these properties, we will call this idealistic component an *oracle*. Depending on the goals and the domain, an oracle can be a reference model or a more permissive abstract contract.

3) Failure traces: From the extended model and the oracle, the model checker will be able to generate a set of traces leading to system-level failures. From each trace, we can extract the values of the trigger variables (i.e., which faults were activated) and the location of the system-level failure detected by the oracle. If the oracle can classify the failure, this information can also be retrieved.

4) Efficiency and Lazy Evaluation: Model checking is highly sensitive to the size and potential values of the state vector. Unfortunately, adding a set of nondeterministic boolean variables (here the trigger variables) increases the number of potential states exponentially. Moreover, if permanent faults are modeled in such a way that the initial activation is random, the number of initial states immediately blows up exponentially.

In order to avoid the combinatoric explosion, we suggest a "lazy" strategy to evaluate fault activations. Let the trigger variables have ternary values, with the third value being *undefined*, also being the initial value. By injecting additional logic to access the value of trigger variables, it is possible to defer the valuation and have identical states for multiple fault configurations up to an actual fault activation. This effect is illustrated in Figure 2.

IV. Evaluation of Fault Tolerance Mechanisms and Error Detectors

Section III outlined a general approach to model checking-based automated SW-FMEA. In this section, we present two novel applications of the method to evaluate fault tolerance mechanisms and error detectors. The goal is to measure the efficiency of these mechanisms by analyzing what type of faults they can mask or detect, respectively.

For an "absolute" measure, one can use an idealistic oracle (like we suggested in Section III-2) as a baseline and "upper bound" on the efficiency of realistic approaches. In case of error detectors, it is also possible to compute the *relative efficiency* of two solutions, i.e., how much "better" or "worse" is one of them compared to the other.

The measurement setting is the following. In case of error detectors, we first run a check with the oracle (or the first detector) to get the total number of traces leading to observable failures (denoted T as *total*), then we measure the same number (denoted D as *detected*) with the evaluated (or second). In case of fault tolerance mechanisms, both steps use the oracle, with the mechanism coupled with the system in the second step. Efficiency is then defined as follows.

- In case of error detectors, the efficiency is $E = D/T$.
- In case of fault tolerance mechanisms, the efficiency is $E = (T - D)/T$.

Efficiency can also be defined in case of fault types (or failure modes), giving a more detailed picture about the evaluated technique. By obtaining a number describing the efficiency of different approaches, we hope to help design decisions concerning what error detectors and fault tolerance mechanisms to use (possibly in some combination).

V. OSEK API – A Case Study

To demonstrate the merits of the proposed approach, we used the model of the OSEK API [1], a common interface definition for safety-critical embedded operating systems. In a related project[1], an OSEK-compliant real-time operating system targeting the automotive industry had to be certified according to ISO 26262 [3]. The developers of the OS wanted to add fault tolerance and monitoring techniques addressing potential programming errors, both from the side of the OS and client applications. To aid design decisions, we have developed the presented approach to evaluate different solutions still in the modeling phase. For the analysis, we used a model of the OSEK API and a set of test programs (both correct and incorrect) taken from [9] and a set of error detectors with assertions providing the "error signals".

The OSEK API provides a set of interface functions with their syntaxes and also the semantics of the implemented OS primitives. The API defines primitives for task handling and scheduling; resource, interrupt and event handling; semaphores and messaging; as well as times and alarms. For the case study, we used a model describing the Task API, the Resource API and the Event API.

We have modeled two types of error detectors: as the idealistic oracle, a (fault-free) Reference Model that is compared to the state of the OSEK model after each interface call (according to the Master-Checker pattern); as well as a simple Priority Checker that observes only the priorities of scheduled tasks. The Priority Checker can detect if the scheduler violates the priorities, for example by preempting a task to run another one with lower priority.

A. Implementation of Automated SW-FMEA

We have implemented the approach described in Section III based on the model checker SPIN. Fault injection was per-

[1]This work has been partially supported by the CECRIS project, FP7–Marie Curie (IAPP) number 324334.

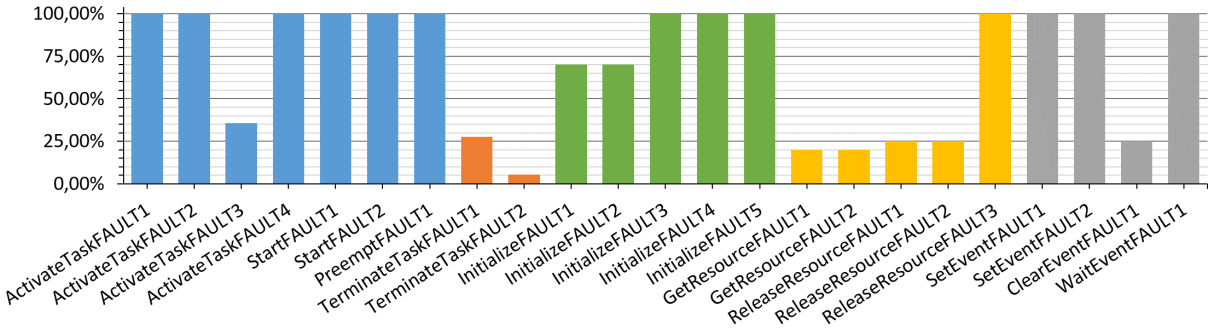

Fig. 3: Efficiency of the Priority Checker compared to the Reference Model

formed by an auxiliary program that parsed the PROMELA model of the OSEK API and altered the code. We used a very simple fault model: each instruction in the model could be removed when activated by a trigger variable. We assumed a single, permanent fault that activates in the initial state.

SPIN was configured to perform a bounded depth-first search optimized for safety checking and enumerating every violating trace. The tool looked for assertion violations (errors detected by the evaluated error detectors) and invalid end states (i.e. deadlocks). Once the model checking finished, the path was replayed to obtain the last (violating) state, containing the values of the trigger variables and the location of the error signal. This information was aggregated for all traces, resulting in the number of violating traces for each different fault-type.

B. Results

Running the analysis with the two detectors showed the relative efficiency of the Priority Checker compared to the more "heavyweight" Reference Model. The diagram in Figure 3 illustrates the efficiency for each fault type (alteration in the API model) separately, also grouping them based on the related API. Although the fault model is artificial, the diagram highlights that the Priority Checker can barely detect faults in the resource handling or task termination primitives, but it is comparable to the Reference Model for most of the faults related to rescheduling (starting tasks and handling events).

In a real world example, analysis of the characteristics of different detectors could help in understanding their efficiency (or coverage) better. In this study, the Reference Model can also be regarded as an idealistic oracle, while something like the Priority Checker can be implemented for an acceptable cost. By knowing the costs of a solution and its characteristics, it should be easier for engineers to find a cost-optimal solution with the highest possible benefits.

VI. Conclusion and Future Work

This paper presented a method for automated SW-FMEA based on model checking, along with a novel idea for applying such approaches in the evaluation of fault-tolerance mechanisms and error detectors.

The main idea of the model checking-based method is to *1)* use model-level fault injection (or model mutations) with trigger variables to augment the system model with switchable faults, then *2)* use formal specification or an oracle model to characterize system-level failures so that *3)* the model checker can generate traces leading from a fault activation to a failure.

Evaluation of fault-tolerance mechanisms and error detectors is based on the notion of (relative) efficiency that describes the number of masked/revealed errors compared to an oracle or another technique (respectively). We hope that this additional piece of information can aid safety-engineers in early design decisions.

The main contribution of this paper is the outline of a general idea. In order to make it applicable, there are a number of further concerns to be considered. First, the fault model for executable software models has a great impact on the validity of the results, so a fine-tuned and validated fault model is necessary. We plan to use completed projects with code-level fault injection to statistically compare the effects of model-level and code-level faults. Secondly, a specific model checking algorithm could inherently optimize the structure of the state space without lazy evaluation injected into the model (see Section III-4). Thirdly, the case study presented here is only in a preliminary phase – modeling other aspects of the OSEK API and additional error detectors or fault-tolerance mechanisms will be necessary to extract meaningful results.

References

[1] Road vehicles – open interface for embedded automotive applications. ISO 17356, 2005.

[2] Potential failure mode and effects analysis in design (design FMEA), potential failure mode and effects analysis in manufacturing and assembly processes (process FMEA). SAE J 1739, 2009.

[3] Road vehicles – functional safety. ISO 26262, 2011.

[4] V. Bonfiglio, L. Montecchi, F. Rossi, P. Lollini, A. Pataricza, and A. Bondavalli. Executable models to support automated software FMEA. In *Proc. High Assurance Systems Engineering*, pages 189–196. IEEE, 2015.

[5] L. Grunske, K. Winter, N. Yatapanage, S. Zafar, and P. A. Lindsay. Experience with fault injection experiments for FMEA. *Software: Practice and Experience*, 41(11):1233–1258, 2011.

[6] G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[7] Yue Jia and M. Harman. An analysis and survey of the development of mutation testing. 37(5):649–678, 2011.

[8] Chris Price and Neal Snooke. An automated software FMEA. In *Proc. International System Safety Regional Conference (ISSRC)*, 2008.

[9] H. Zhang, T. Aoki, and Y. Chiba. A SPIN-based approach for checking OSEK/VDX applications. In *Formal Techniques for Safety-Critical Systems*, volume 476 of *CCIS*, pages 239–255. Springer, 2015.