Automated Freedom from Interference Analysis for Automotive Software

Florian Leitner-Fischer
ZF TRW
78315 Radolfzell, Germany
Email: florian.leitner-fischer@zf.com

Stefan Leue
Chair for Software and
Systems Engineering
University of Konstanz
78457 Konstanz, Germany
Email: stefan.leue@uni-konstanz.de

Sirui Liu
Department of Computer and
Information Science
University of Konstanz
78457 Konstanz, Germany
Email: sirui.liu@uni-konstanz.de

***Abstract*—Freedom from Interference for automotive software systems developed according to the ISO 26262 standard means that a fault in a less safety critical software component will not lead to a fault in a more safety critical component. It is an important concern in the realm of functional safety for automotive systems. We present an automated method for the analysis of concurrency-related interferences based on the QuantUM approach and tool that we have previously developed. We illustrate the proposed analysis using the case study of an airbag system.**

I. Introduction

The identification of interferences amongst system components is an important aspect in safety analysis. It contributes to fault containment and avoidance by ensuring that a fault in one system component does not lead to a fault in another component.

When considering software driven embedded systems, such as automotive ECUs, interference can happen at various levels. The software of one component can access and manipulate another component by writing faulty data into the memory allocated to the other component (data flow). One component can even change the control of another component by manipulating the runtime stack (control flow interference). Finally, one component can stop to send information to another component, or even worse, it can send faulty information to another component (message flow, byzantine faults). When considering concurrently executing software components, the faulty behavior of one component may cause the system to go into a deadlock. Likewise, one component executing with high priority may starve other components at lower priorities. This is particularly damaging if the higher priority component goes into a livelock that is not preempted by the operating system scheduler. Finally, various damaging scenarios can occur if the execution of a task in a component leads to exceeding real-time bounds in another component.

The investigation of these types of interference is of particular importance in the safety analysis of safety-critical automotive software. The standard ISO 26262 [1] defines various Automotive Safety Integrity Levels (ASILs) for different safety-critical functions implemented in an automobile, ranging from A to D, with ASIL D being the ASIL that corresponds to the highest criticality. ISO 26262 defines freedom from interference (FFI) as the "absence of cascading failures between components that could lead to the violation of [some] safety requirement" (c.f. part 6 of [1]).

Special interest in establishing FFI for automotive systems developed according to 26262 is motivated by the fact that the standard allows functions assessed at a higher ASIL level to be implemented by redundant functions developed according to development methods applicable to lower ASIL levels, provided there is no interference and dependence amongst these lower level implemented functions. This is referred to as "ASIL decomposition". Performing ASIL decomposition is attractive since the development process requirements that 26262 imposes on lower criticality ASILs are lower than on higher level ASILs. This has a significant impact on the overall development cost of some function: the total development cost for n redundant lower ASIL components may be less than the cost for developing one high ASIL component, while still ensuring high dependability of the resulting system.

Various techniques can be employed that will ensure FFI. Most prominent in this regard is the use of Memory Protection Units (MPUs) which ensure at *run-time* that a software task is only accessing memory that was allocated to it. Contrary to this approach, we are proposing a *design-time* analysis method that allows the designer to avoid certain types of interference before the product is manufactured and put into operation.

In light of ever increasing system complexity, manual analysis methods to ensure FFI will not succeed in the long run. Testing is not applicable, since the concurrency-related interference problems such as deadlock and livelock that we will focus on in this paper cannot be effectively tested. Also, testing is incomplete, making it a questionable method in functional safety analysis. We therefore propose an automated, algorithmic analysis method to establish the absence of interferences based on the formal methods of model- and causality checking [2], [3]. The analysis is based on our previously developed QuantUM approach and tool. By this approach we are able to directly access architecture-level SysML / UML models given by some industry-strength modeling tool, such as IBM Rational Rhapsody, and analyze these models for interference violations without the need for user intervention. We show how to capture FFI analysis in this setting and how

to perform the analysis. We illustrate our approach using a case study and discuss future developments.

Related Work: We are not aware of any comparable, formal methods based FFI analysis method that focuses on concurrency faults.

II. Preliminaries

A. The QuantUM Approach

In precursory work [4], [5], [3] we have proposed the QuantUM approach to support the automated, algorithmic functional safety analysis of critical system architectures. QuantUM analyses are based on SysML [6] system architecture models. Designers will provide models for the structure (block definition diagrams, internal block diagrams) as well as the normal and the fault behavior in the form of SysML StateChart diagrams. We provide an extension to SysML in the form of a stereotype that allows the designer, amongst other things, to distinguish normal and faulty behavior and to specify failure rates for individual architectural blocks. The thus extended SysML models are edited in a SysML tool, such as IBM Rational Rhapsody, and their XMI representation is then parsed and input by the prototypical QuantUM[1] tool that we have developed. QuantUM uses various model checking [2] tools in order to perform a causality analysis, that we refer to as causality checking [3], in order to compute ordered sequences of events that lead to the violation of a safety goal. Model checking is an automated, algorithmic technique to systematically explore the state space of the system, i.e., all possible configurations of the system, in order to discover the reachability of undesired system states. We have performed various large-scale case studies using this tool suite [7].

B. FFI According to ISO 26262

Space limitations will not permit us to fully explore the intricate relationships of cascading faults, interference and independence in ISO 26262. The informative Annex D of part 6 of ISO 26262 lists in clause D.2.2 ("Timing and Execution") the following types of faults to be considered for software elements considered in each software partition: a) blocking of execution, b) deadlocks, c) livelocks, d) incorrect allocation of execution time and e) incorrect synchronization between software elements. The notion of concurrency does not occur in this context, and no examples for these faults are given, but we assume that fault types b), c) and e) are directly related to concurrency issues. Fault type e) probably refers to what is typically considered to be race conditions in concurrent systems. The technology of model and causality checking that we employ in our approach is very well suited for the analysis of these types of concurrency problems, which is why we will focus on this important aspect of FFI analysis. Notice that according to part 8 of ISO 26262, clause 9.4.1.1, model checking is an admissible method for verification.

[1]https://se.uni-konstanz.de/research1/quantum/

III. FFI Analysis using QuantUM

The FFI analysis method that we propose in this paper will focus on concurrency issues that potentially entail cascading faults. The analysis will involve the following steps:

1) *System Modeling:* The normal behavior of every system component will be modeled in SysML using state machine diagrams. Emphasis should be put on modeling the chain of data transfer and communication between components in a system.
2) *Fault Seeding:* This step involves identifying system components that are involved in the occurrence of a concurrency fault. The considered fault needs to be seeded in the model which involves modifying the previously obtained system model.
3) *Fault State Identification & QuantUM Tagging:* A state of the overall system needs to be identified that corresponds to a violation of the safety goal of the system. Using the QuantUM provided stereotype the system components that the analysis is supposed to consider will be tagged. At the same time, the normal and the failure behavior will be identified.
4) *QuantUM FFI Analysis:* The subsequent analysis of the model using the QuantUM tool is fully automatic. The result will be a fault tree displaying ordered sequences of events that are causal for the occurrence of the identified concurrency issue.
5) *Result Interpretation:* The final step will be an interpretation of the obtained result. This is of course to some extent specific to the property that is to be analyzed, but in principle proceeds as follows: If an event of a component with lower ASIL occurs in the ordered set of causal events prior to an event with higher ASIL, this is an indication for an interference.

IV. Case Study

A. Background

On the one hand, a deadlocks within a component causes interference on other components that depend on that locked component, it may remain questionable whether an interference analysis is necessary for deadlocks or livelocks. Deadlocks and livelocks are caused by software errors and consequently, according to the ISO 26262 philosophy, they could be detected and corrected at software design time. In fact, tools and techniques exist to discover possible deadlocks in a design model of a software system [8]. It is hence questionable whether these errors should be subject to FFI analysis at all. On the other hand, we assume that, in real world scenarios, the system is not free from software errors. Deadlocks and livelocks then will still exist in the system, so it still remains our interest to find out whether potential errors of this kind in subsystems can be source of a violation of FFI.

We consider the abstract architecture of an Airbag Control Unit. We are interested in analyzing what impact the failure of two redundant decision paths executing on one micro controller has on achieving the safety goal of the overall system. The system safety goal is to avoid unintended deployment, i.e.,

the case where no crash occurs but the airbag is deployed.If such an impact can be proven, we have shown the presence of a cascading failure, which implies interference. Since the QuantUM analysis that we perform is complete, which means that all possible system executions will be explored, we can infer from the absence of such an impact that FFI holds for the considered safety goal, concurrency fault and system model. Notice that the two decision paths could be the result of some ASIL decomposition, which would be justified in case our analysis proved FFI.

Prior to performing this case study, we carried out an analysis of the interference that could be caused by the subsystem consisting of the two decision paths going into a mutual deadlock. Although the result seems to be trivial, given that a (sub-)system is not able to perform any operations when all its components are in a deadlock, we were able to show this behavior analytically using QuantUM analysis as follows. We modeled the normal behavior pattern of the system in such a way that it was impossible for the two decision paths to end up in a deadlock, as well as a failure pattern, where the system will surely end up in a deadlock. The resulting fault tree with the deployment of the airbag as top level event did only show events of the normal operation, no events in the failure pattern were found in the fault tree, which indicates that a behavior pattern where all the deciding components will surely end up in a deadlock can not lead to a deployment of the airbag. On the other hand, by removing the normal behavior pattern, it was not possible to deploy the airbag. This analysis gave us an demonstration on how QuantUM can show us that the first safety goal is not violated by a deadlock of the two decision paths. Knowing that a deadlock will not lead to a deployment, the next question was whether we can introduce a safety mechanism that ensures the second safety goal that the airbag should deploy when a crash occurs. In this case study, we adapt the behavior pattern of the two decision paths in such a way that they can lead to a deadlock and introduce a watchdog module that detects whether the two decision paths are currently in a deadlock condition, resetting them if necessary.

B. Freedom From Interference Analysis

1) System Modeling: We begin by modeling the normal behavior of the decision paths and consider the main path in our description. The ”safing” path would be modeled in a similar fashion. As shown in Figure 1, upper part, the behavior of the decision path is such that by default at the system start the decision path goes to the idle state. When a crash is detected, the considered decision path checks whether the other decision path also detected a crash. No matter whether there is a response from the other path, or what that response is, a deploy signal is sent out. Benefits at this level of abstraction is that we do not need to consider modeling the communication between the two decision paths since it does not have an influence on the emission of the deploy signal. In a real implementation the response from the other decision path could for instance be used to log an error in case not both decision paths compute the same result.

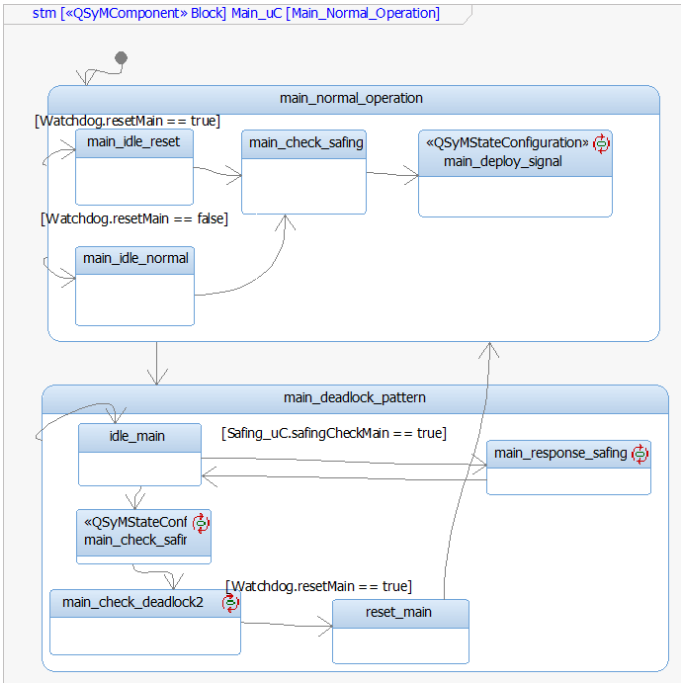

Fig. 1. Behavior of the System

2) Fault Seeding: We continue by seeding a deadlock into the model. In a real design situation this may, for instance, be caused by a coding error. As seen in Figure 1, lower part, similar to the normal behavior the decision path enters the idle state when the system starts. In the idle state, there are two possibilities. First, after a crash is detected, the considered decision path enters the state where it communicates with the other decision path. The difference to the non-seeded model is that the decision path cannot proceed with its computation until it received a response from the other decision path containing information about whether it also detected a crash. When both decision paths enter this state, a circular wait emerges between them, which indicates a deadlock between the two decision paths. Second, the considered decision path may receive a signal requesting a ”crash detected” response from the other decision path (refer to the first possibility). In this case, the decision path that received the request, enters the response state. In this state, it sends out a response message, containing information about whether it also detected a crash, to the other decision path.

Knowing that there is a seeded interference error present, we want to check whether there is a way to deploy the airbag with the introduced safety mechanism. To check this, we model the behavior of the airbag. As shown in Figure 2, the airbag can enter a ready state when one decision path sends out a deploy signal. It must receive a deploy signal from the other decision path to actually deploy.

A watchdog (Figure 3) will monitor the two decision paths. Since we know that there is a deadlock deliberately seeded in our model, we only need the watchdog to check whether our system entered the state where the two decision paths are in

the seeded deadlock state and reset the two decision paths if necessary.

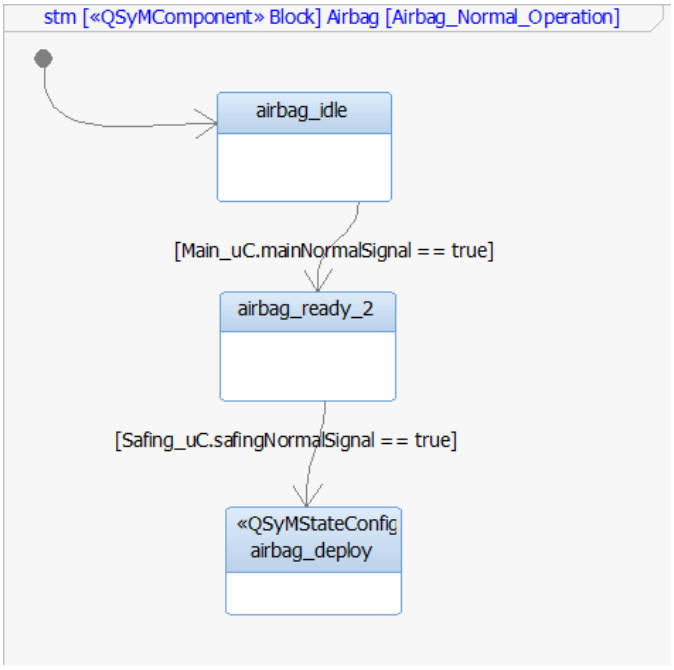

Fig. 2. Behavior of the Airbag

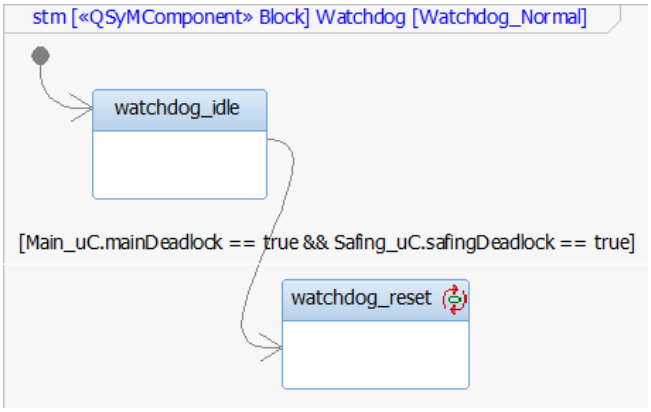

Fig. 3. Behavior of the Watchdog

3) Fault State Identification & QuantUM Tagging: It has to be determined which state of the overall system we want to analyze. We choose this state according to the safety requirements of the system. In this case, it is the deploy state of the airbag module, which is shown in Figure 2. Consequently, we annotate the airbag deploy state in the airbag module as a critical state, and pass the annotated model in XMI format to the QuantUM tool.

4) QuantUM FFI Analysis: QuantUM analyzes the SysML model and returns the fault tree depicted in Figure 4. The analysis took about one minute of execution time on a computer with a Core 2 Duo CPU@2.26GHz and 4GB RAM.

5) Result Interpretation: The QuantUM tool automatically generates fault trees that visualize ordered sequences of events that are causal for the occurrence of a top level event. We slightly abuse the fault tree formalism here in order to visualize causal events that lead to the top level event representing the deployment of the airbag, which does of course not correspond to a hazard but to an event whose occurrence illustrates the satisfaction of the safety goal that we pursue. The fault tree, which is reproduced in Figure 4, shows that there are two situations that can lead to an airbag deployment. First, the left part of the fault tree shows a sequence of exclusively normal operation events, this indicates that the airbag can deploy when functioning normally. Secondly, the right part of the fault tree shows a sequence of the two decision paths going into a mutual deadlock which is then reset by the watchdog. After the reset, a sequence of normal operations follows, leading into the airbag_deploy state, which corresponds to the top level event of the fault tree. From this we conclude that the watchdog we introduced ensures deployment of the airbag even though a deadlock was present. Hence, we have shown the effectiveness of the introduced safety mechanism.

V. Conclusion

We have presented an automated approach to FFI analysis based on the QuantUM analysis method and tool that we have previously developed. We have applied this approach to FFI analysis of a Airbag Control Unit case study. We have shown that the technology that we have used, in particular model and causality checking, are suited to analyze FFI for concurrency-related faults, such as deadlock. Future research will address the question how to automate the fault seeding, so that the analysis can be performed fully automatically. A further objective will be to incorporate the analysis into standard system and safety design processes. It is our impression that more discussion in the ISO 26262 standardization body is required in order to clearly define and characterize the fault types that are to be considered with respect to FFI, in particular as far as clauses D.2.2 and D.2.4 of part 6 of the standard are concerned. We see great potential in using formal methods to characterize these properties and to provide automated, formal methods based FFI analyses for them.

References

[1] ISO, "Road vehicles - functional safety," International Organization for Standardization, Geneva, Switzerland, ISO 26262, 2011.
[2] C. Baier, J.-P. Katoen *et al.*, *Principles of model checking*. MIT press Cambridge, 2008, vol. 26202649.
[3] F. Leitner-Fischer and S. Leue, "Causality checking for complex system models," in *VMCAI*, ser. Lecture Notes in Computer Science, vol. 7737. Springer, 2013, pp. 248–267.
[4] ——, "Quantum: Quantitative safety analysis of UML models," in *QAPL*, ser. EPTCS, vol. 57, 2011, pp. 16–30.
[5] ——, "Probabilistic fault tree synthesis using causality computation," *IJCCBS*, vol. 4, no. 2, pp. 119–143, 2013. [Online]. Available: http://dx.doi.org/10.1504/IJCCBS.2013.056492
[6] OMG, "Omg systems modeling language," OMG, Tech. Rep., 2015. [Online]. Available: http://www.omg.org/spec/SysML/1.4/PDF/
[7] F. Leitner-Fischer and S. Leue, "Spincause: a tool for causality checking," in *SPIN*. ACM, 2014, pp. 117–120.
[8] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Addision–Wesley, 2003.

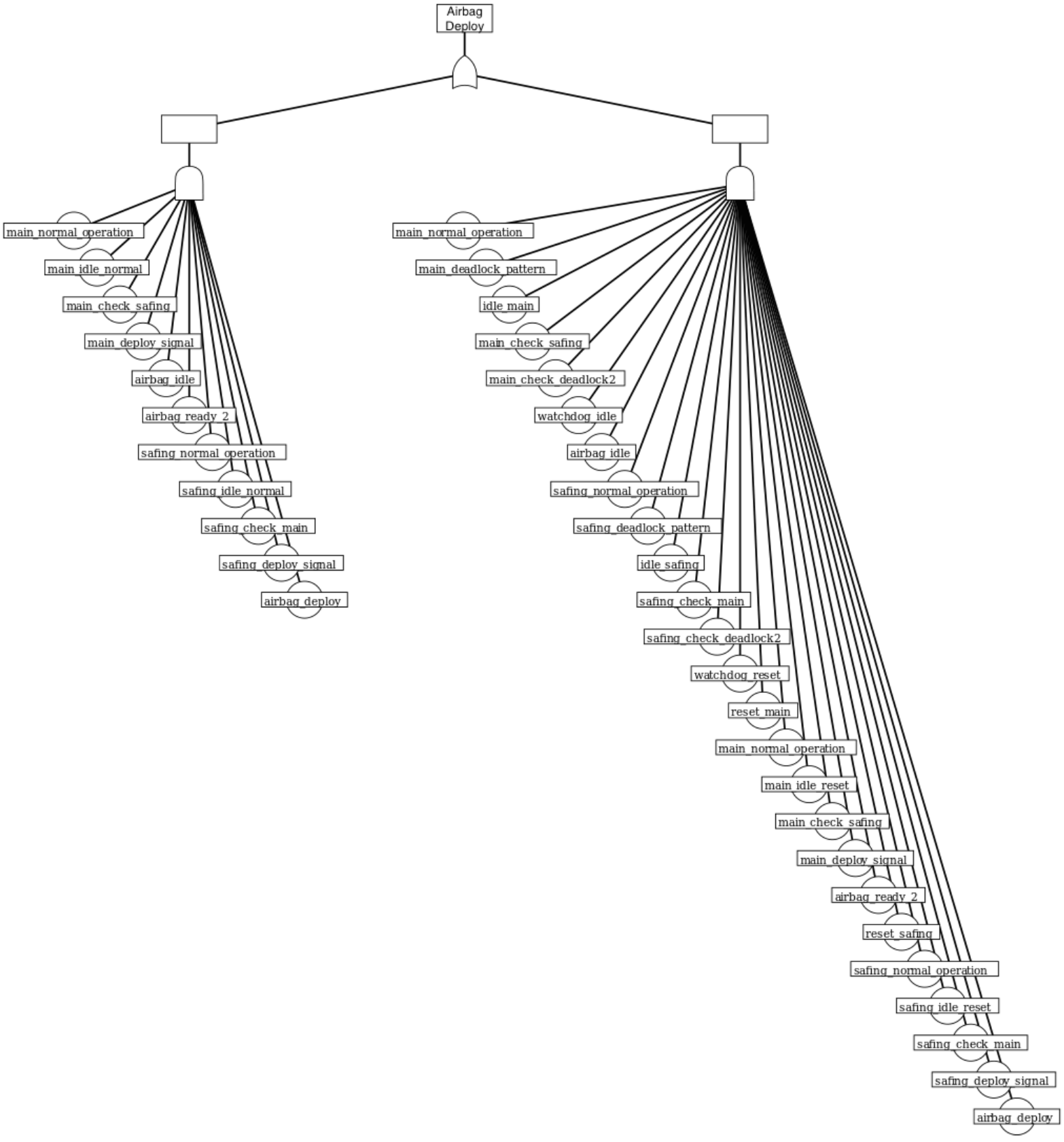

Fig. 4. Fault Tree of the deployment of the Airbag