A Generic Method for a Bottom-Up ASIL Decomposition

Alessandro Frigerio[1], Bart Vermeulen[2], and Kees Goossens[1]

[1] Eindhoven University of Technology, The Netherlands
[2] NXP Semiconductors, Eindhoven, The Netherlands
a.frigerio@tue.nl

Abstract. Automotive Safety Integrity Level (ASIL) decomposition is a technique presented in the ISO 26262: Road Vehicles - Functional Safety standard. Its purpose is to satisfy safety-critical requirements by decomposing them into less critical ones. This procedure requires a system-level validation, and the elements of the architecture to which the decomposed requirements are allocated must be analyzed in terms of Common-Cause Faults (CCF). In this work, we present a generic method for a bottom-up ASIL decomposition, which can be used during the development of a new product. The system architecture is described in a three-layer model, from which fault trees are generated, formed by the *application*, *resource*, and *physical* layers and their mappings. A CCF analysis is performed on the fault trees to verify the absence of possible common faults between the redundant elements and to validate the ASIL decomposition.

Keywords: ADAS, ASIL decomposition, automotive architecture, common-cause fault analysis, fault trees, functional safety, ISO 26262

1 Introduction

Automotive Safety Integrity Level (ASIL) decomposition is a standardized practice presented in *ISO 26262: Road Vehicles - Functional Safety* [8]. This technique is used to reduce the criticality of safety requirements. It is generally applied during the allocation of the ASIL values to the safety requirements. The ASIL value of a requirement corresponds to a minimum ASIL that the system, which consists of a given mapping of applications, resources, and locations, must be able to achieve. When sufficiently independent architectural elements are present, the safety requirements can be split into less critical ones and mapped to the independent elements.

Figure 1 shows an example of a simple application and its mapping to the resources (1a) and a corresponding version in which the processing part *proc* is implemented by two different functional nodes, *proc1* and *proc2*, and executed by different processors, *ecu1* and *ecu2* (1b). The *split* and *merge* nodes provide the safety mechanisms to obtain the correct application functionality with high reliability. They are implemented in this example by the sensor and the actuator respectively. The application layer contains the ASIL related to a safety

requirement, while the resource layer has ASIL specifications that must satisfy the application requirements. The implementation resources are then mapped to the physical layer.

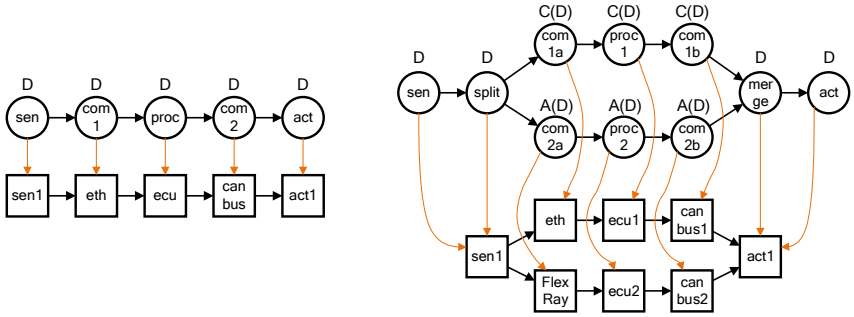

Fig. 1. Example application mapped on a resource graph (1a) with redundancy (1b). The ASIL requirements for the application are shown above the application nodes. The notation X(Y) refers to a decomposed requirement in which X is the new value and Y the original.

To validate the ASIL decomposition shown in Figure 1, according to the ISO 26262 standard, the redundant elements must be independent, meaning that they cannot have Common-Cause Faults (CCFs) that could result in a system failure [8]. The independence must be analyzed in terms of software and hardware design and implementation, failures of adjacent elements, environmental factors, failure of common external resources, etc. The ASIL decomposition can be approved only after the analysis of the CCFs.

In this paper we approach the ASIL decomposition in a bottom-up fashion. Compared to a top-down approach, where the ASIL requirements are allocated to an existing architecture, we modify the architecture introducing independent elements on which the redundant requirements can be allocated.

To this end, we present a three-layer model of automotive Electrical and Electronic (E/E) architectures. It is used to analyze the system with automated tools, validate the ASIL requirements from the mapping of the applications, and introduce system redundancy by modifying the structure of the architecture. From the architecture model we generate fault trees for each application that is executed in the vehicle. We use application, resource, and physical space model to analyze the independence of the redundant parts of the system. A model implemented since the early stages of the development phase helps the system architects to maintain proper documentation and to trace the requirements on the implementation. When comparing different solutions, a model-based approach helps in making the trade-offs between safety, availability of the products, costs, and performance of the implementation.

An inspection of the fault trees allows the detection of a CCF that will cause breaks in the modules's independence assumptions, exposing the situations in which the ASIL decomposition would not be valid.

The novel contribution of this work is:

- A three-layer model that consists of *application*, *resource*, *physical* layers, and their mappings, that explicitly expresses redundancy with specific application and resources elements, to perform the ASIL tailoring process on the implementation level;
- Automated fault trees generation from the model, which are used in the CCF analysis. This validates the independence of redundant elements, as required by the ASIL decomposition process described in the ISO 26262 standard;
- Model transformations to modify the degree of redundancy of the system and lower the ASIL requirements for single elements, while maintaining the ASIL of the system as a whole.

The rest of the paper is organized as follows: Section 2 provides an overview of the ISO 26262 safety standard. Section 3 describes the architecture model that is used in this work, and Section 4 discusses redundancy in terms of model transformations. Section 5 introduces fault trees and the generation algorithm to synthesize them from the architecture model. Section 6 presents the related work and Section 7 concludes the paper by summarizing our results.

2 ISO 26262: Road vehicles - Functional Safety

The ISO 26262: *Road Vehicles - Functional Safety* standard, published in 2011, addresses the safety aspects of automotive E/E architectures, considering both random and systematic system failures. It is an automotive-specific adaptation of the IEC 61508 standard [7], which focuses on functional safety of general electronic systems.

The ISO 26262 standard is divided into 10 parts, analyzing safety requirements during all the product life-cycle. It provides guidelines on the management of safety requirements, as well as which safety requirements are necessary for the concept phase of the product, its hardware and software development, the production and the validation of the system. Moreover, it provides guidance on ASIL-oriented requirements and decomposition. A second edition of the standard will be published in 2018 focusing on motorbikes and providing guidance on the application of the standard and ASIL definition to hardware components.

2.1 Automotive Safety and Integrity Level

Safety can be measured with the Automotive Safety Integrity Level (ASIL) concept, which is similar to the Safety Integrity Level (SIL) of IEC 61508. The ASIL system uses a risk-based approach that takes into account the *Severity, Exposure,* and *Controllability* of a potential harm. There are five possible levels:

from the most critical ASIL D to the least critical ASIL A and a QM (quality management) level that refers to non-safety-critical items. Figure 2 shows how the ASIL values are calculated based on the three risk parameters. The highest level D corresponds to all the risk parameters being at their maximum: S3 corresponds to life-threatening or fatal injuries, E4 to a high probability of exposure and C3 to a difficult to control or uncontrollable risk.

2.2 Requirement Decomposition

The ISO 26262 standard "*provide(s) rules and guidance for decomposing the safety requirements into redundant safety requirements to allow ASIL tailoring at the next level of detail*" [8]. Lower ASIL requirements for the implementation resources on which the application nodes are mapped on could be necessary for three main reasons during the product development:

1. Elements with the maximum criticality level are not available. Creating ASIL D compliant devices is a difficult task, and often the highest safety level can be achieved only by exploiting the knowledge of the application that the device will support. This is not possible for general purpose elements or resources that are shared by many applications.
2. High-ASIL software is difficult and expensive to develop and test. The same holds for the software development tools used;
3. The production process used to create a safety-critical component is expensive. Decomposing the system into less-critical elements may be the most cost-efficient solution.

		C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

Fig. 2. ASIL determination table

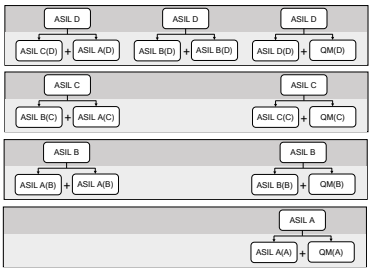

Fig. 3. Possible decomposition schemes

Figure 3 shows the acceptable ASIL decomposition schemes defined by the standard, which follow the rule of Equation 1. To the ASIL values, QM to D, we assign a number from 0 to 4, and the following relation must be satisfied:

$$ASIL_{orig} \leq \sum ASIL_{decomp} \tag{1}$$

The standard uses the notation $ASIL_{decomp}(ASIL_{orig})$ to mark which elements have been decomposed and trace the original requirement. Additional

procedures must be carried out when decomposing ASIL D requirements, for example, the test and the integration of each decomposed element shall be implemented in compliance with ASIL C. In particular, when a requirement is decomposed into redundant elements, it is necessary to establish independence between them for the original safety requirement to be correctly satisfied. For example, the redundant elements should not depend on a common resource, such as a shared battery, that could cause them to fail simultaneously, i.e. a CCF.

3 Three-Layer Architecture Model

When describing Advanced Driving Assistance Sytems (ADAS) or Autonomous Driving (AD) related applications, the *sense-think-act* paradigm is generally used. It is a common concept used in Robotics, which separates an application into three main domains:

a) *Sense:* an application will always start by collecting information about the surrounding environment or the vehicle status from one or more sensors.
b) *Think:* the collected data is then processed. Different design approaches can be used to determine if it will happen, for example, in a centralized architecture, where a single module will analyze the data, or in a distributed fashion, in which multiple modules will analyze the different sensor data.
c) *Act:* the final part of an application involves the actuators, which modify the status of the vehicle.

In this work we assume that all applications follow this paradigm, and in the application graph a path from each actuator to at least one sensor always exists.

3.1 Model Description

Modeling the automotive E/E architecture is necessary to analyze the system. To validate ASIL decomposition it is necessary to include both the descriptions of the applications, the implementation resources used, and the physical space of the vehicle.

A three-layer approach is used: the architecture is described in terms of *application, resource*, and *physical* layers. The *application* layer contains disjoint application graphs, while the *resource* and *physical* layers contain one graph.

The *application layer* can contain multiple graphs, each describing a different application. Each application is related to a specific safety requirement, for example availability of the system for a certain task, derived from the safety goals, analyzed during the Hazard Assessment and Risk Analysis (HARA) phase. The *application layer* describes the functional architecture of the vehicle by defining the relationships between the software nodes via a directed cyclic graph $G = (N, E)$ for each application, where N is the set of software nodes and E is the set of edges that connect the nodes. Each node has an ASIL requirement, which is originally inherited from the initial safety requirement, but can be lowered by the ASIL decomposition procedure. The edges indicate information flow

between the nodes, but do not have any capacity or timing properties, which are expressed by explicit communication nodes. Each node has a specific type:

a) *Functional:* the computational aspects of an application;
b) *Communication:* the communication aspects of an application;
c) *Sensor:* the data source of an application;
d) *Actuator:* the data sink of an application;
e) *Splitter:* node that replicates the input data to its output ports;
f) *Merger:* node that compares the redundant inputs and ensures only correct outputs are forwarded.

Note that the *splitter* and *merger* nodes are necessary to describe the redundant elements of the system, and will be discussed in the following sections.

The *resource layer* describes the implementation architecture of the vehicle, comprising of hardware and software elements. The resources are expressed with a directed cyclic graph $H = (R, L)$, in which R is the set of resources and L is the set of links that connect them. Each resource can provide multiple types e.g.:

a) *Functional:* a resource on which the application functional nodes can be mapped on, like a processor or a controller;
b) *Communication:* resources that represent the different types of automotive networks (LIN, CAN, FlexRay, MOST, Ethernet) or direct connections;
c) *Sensor:* a resource that collects data, like a camera or a wireless receiver;
d) *Actuator:* a resource that interacts with the physical environment by executing the desired operations, for example the braking actuator;
e) *Splitter:* a resource capable of forwarding the data received on an input ports to multiple output ports;
f) *Merger:* a resource capable of deciding which input data is correct and forwards it to its output ports.

We model generic resource-resource dependencies in the *resource* layer. To show one example, we use the power supply, but any other shared resource can be modeled in the same way and included in the CCF analysis.

g) *Power Source:* a resource that provides the power supply for other resources, for example a battery;
h) *Power Line:* a resource that distributes the power supply to other resources.

Each resource has a set of types, and the application nodes with that type can be mapped on that resource. Hybrid resources can be described properly by the model, for example a gateway would be both a *Functional* and a *Communication* resource, and might have *Splitter* or *Merger* capabilities too. Each resource has an ASIL value, representing the maximum ASIL value that it can satisfy for a specific safety requirement, usually referred to as a ASIL-X ready resource.

The *physical layer* is described similarly by a cyclic graph $F = (P, C)$, where P is the set of physical locations and C the set of connections. The description

of the physical space is inspired by [10], in which the authors focus on the study of the wiring costs in an E/E architecture and they model the system to analyze wire routing and splice allocations. In our proposed approach, the physical locations can describe: the areas of the vehicle in which the ECUs and hardware components can be placed, which have a limited available space, and the paths in which the communication wires can be positioned and their length.

The interconnections between the different graphs show the mapping of the applications to the hardware resources, and of the hardware resources to the physical locations of the vehicle. Figure 1 is an example of the first two graphs and the relationships between them; it does not show the physical graph to which the resources would be mapped.

4 Model Transformations

Reducing the criticality of each module as much as possible apparently lowers the cost of the product while maintaining high safety. In practice more complications are introduced in the design: safety mechanisms must ensure that the proper functionality is preserved, new communication interfaces are added to the architecture, and a system-level analysis must be performed to ensure that the redundant elements are sufficiently independent.

As a base example, Figure 4 shows the new elements that are introduced in the architecture after the duplication of a single node n, which has only one input and one output for simplicity's sake:

- n_s has a *splitter* type, it collects the inputs and redirects them to the redundant paths;
- c_{1a} and c_{1b} are the new communication nodes that describe the channels between the splitter and the functional nodes;
- n_1 and n_2 are the redundant functional nodes;
- c_{2a} and c_{2b} are the new communication nodes that describe the channels between the two functional nodes and the merger;
- n_m has a *merger* type, it checks the input correctness of the data from the redundant paths and forwards only correct data.

Both the *splitter* and the *merger* nodes are single points of failure for the applications, which means that they will be safety-critical elements that must have at least the same ASIL requirements as the original node n. They perform generic operations on the inputs and outputs of the redundant blocks, for example a merger could be a comparator of a classic k-out-of-n model [1] or part of an health monitoring system which decides which output to use. The other elements of the two branches instead follow the rule presented in Equation 1.

A transformation of the resources can be applied in a similar way. High reliable *splitter* and a *merger* resources will manage the redundant independent resources. The replicated application nodes do not always have a one-to-one relationship with the transformed resources. In a bottom-up approach, the designer

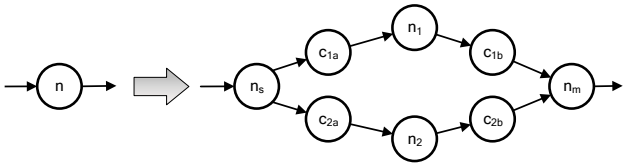

Fig. 4. Node duplication

will make this kind of transformations to the applications and resources layers to create redundant architectures.

Even from a simple transformation, it is clear that replication will introduce a lot of complexity in the system. From a single safety-critical node we introduce at least two nodes, the *splitter* and the *merger*, with the same ASIL value as the original one. In this case, since their functionality is very specific, it is possible to obtain these elements with contained costs compared to a more generic safety-critical one. Moreover, new connections are created and additional latency is introduced by the extra communication and the splitter and merger functionality. New constraints for the design are introduced to meet the independence requirements: redundant application nodes must be mapped on independent resources, and independent resources must be positioned in independent locations. In this work we consider only the safety aspect of these modifications.

5 Common-Cause Fault Analysis for ASIL validation

The information provided by the *application*, *resource*, and *physical* layers allows us to compute the ASIL value obtained with the implementation of each application. If the obtained value is lower than the requirements, it means that the resources cannot satisfy them, and either a different mapping or a different implementation must be used.

The computed ASIL value requires a Common-Cause Fault analysis performed on the three layers of the model to be valid. This analysis can either be manual or automated.

In this work, we generate a fault tree for each application, which is used for an automatic CCF analysis. This analysis recognizes redundancy in the model by searching for *splitter-merger* combinations, and uses the nodes and resources dependencies to determine any possible CCF.

This analysis can also be used to validate a new model after a transformation.

5.1 Fault Trees in Automotive Systems

Fault Tree Analysis (FTA) is a common top-down Safety Analysis, in which an undesired top-level event is identified and then its causes are considered.

In this work we consider as the top-level event the failure of an application for a specific safety goal, for example the system availability, that manifests itself through the failure of at least one of the actuators. Each node, starting from the actuators, can fail because of different reasons:

- Internal failure of the hardware resource on which the application node is mapped or of the software component that implements the functionality;
- Failure of the location on which the used resource is mapped;
- Dependent resource failure. The resource on which the application node is mapped may depend on other resources, such as the power supply;
- Input Failure. Failure of node A that provides data to node B leads to the failure of the node B.

Figure 5 shows the fault tree generated for each node. The same structure is generated for each input application node, until the final sensors are reached, according to the assumption of *sense-think-act* applications. This type of fault tree is based on [6], in which the authors use Dynamic Fault Trees to describe ADAS related applications and perform a FTA.

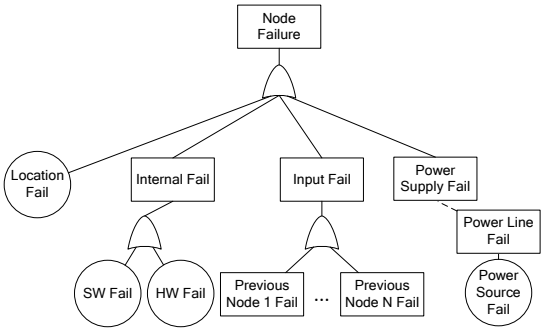

Fig. 5. Subtree for each application node

The internal failure base event could be further developed as in [6], where the hardware and the safety mechanisms implemented in the resources are considered.

5.2 Fault Tree Generation

The fault tree generation algorithm is based on [11]. We assume that the failure of a safety requirement corresponds to the failure of at least one of the related application's actuators. All the actuators are assumed to have the same importance for the success of each application. Assuming a *sense-think-act* paradigm, we can always expect to find a path from an actuator to a sensor.

Algorithm 1 accepts an application graph G and the top-level event e_T as inputs, and then calls the recursive procedure *DevelopSubTree* for each of the application actuators. The function *MappedResource* returns the resource on which an application node is mapped, while the function *MappedLocation* returns the physical location on which the resource is positioned.

For each node, a top fault event is created, to which the four possible fault events described previously in this section are connected via an OR gate. The function *Predecessor* finds all the inputs of an element in its graph. It is used to find all the inputs of the current application graph node, for which a new sub-tree will be generated with the *DevelopSubTree* procedure and connected to the input failure event of the parent node. In case of a *merger* type node, this connection is made through an AND gate, meaning that an input failure is acquired only when all the different input branches of a redundant part of an application fail. In case of a *sensor* type node, no input nodes can be found, the input failure event is deleted and the sub-tree is returned. For all the other nodes, a failure of any of its input leads to a fault, so they are connected with an OR gate.

The power supply event is developed in the *DevelopResourceSubTree* procedure, which is similar to *DevelopSubTree*, but works on the resource and physical layers only. This procedure is generic for resource-resource dependencies, and in this example we use it to generate the power supply fault tree. Each resource can be connected to one or more power source via power lines, which are modeled as resources in the graph. This recursive function travels through the graph, from a resource to each of its power supply, instantiating three types of events: a fault in the power line or power source resource, a fault in the physical location and a fault from the parent power supply resource.

Each element of the fault tree graph is related to the relevant architecture node. The fault trees are saved as graphs, but also exported in the text based Galileo format [5], which is a generic format supported by commercial FTA tools. By adding to the graph information related to the fault rates of each element and the fault probabilities of external events, it is possible to analyze them with the commercial tools and compute reliability metrics for the design.

5.3 Example Scenarios

In this section we discuss three possible scenarios in which our analysis can provide important information to the system architect. In Figure 6a we see the application showed previously in Figure 1b, its mapping to the hardware resources and their positioning in the physical space. The redundant communication resources are correctly placed in different parts of the vehicle so that they will not suffer from CCF related to the environment, while *ecu1* and *ecu2* are placed in the same location *f2*. This is an example in which a CCF related to a common location is found, and a warning is issued to the designer by our analysis tools. Note that in this situation, *proc1* and *proc2* are two different function. If, for example, the function *proc1* was used in both redundant branches, then the possibility of a CCF derived from a systematic fault in the function would have been highlighted by the tool.

A second scenario in which a CCF is found is shown in Figure 6c. Two different power supplies are used for the redundant paths of the first example. Since both the power lines are connected to elements of both the branches, a

Algorithm 1 Fault Tree Generation
Inputs: Application graph G, Resource graph H, Physical graph P, Top-Level Event e_T
Output: Fault Tree F

```
procedure GenerateFT(G, e_T)
    for n_i ∈ N s.t. NodeType(n_i) = actuator do
        e_i = DevelopSubTree(n_i)
    CreateGateOR(e_T, ∀e_i)
procedure DevelopSubTree(n)
    F_sub = CreateNodeEvent(n)
    r = MappedResource(n)
    p = MappedLocation(r)
    e_1 = CreateResourceBasicEvent(r)
    if NodeType(n) != sensor  then e_2 = CreateInputFaultEvent(n)
    e_3 = CreatePowerSupplyEvent(r)
    e_4 CreateLocationBasicEvent(p)
    CreateGateOR(F_sub, (e_1, e_2, e_3, e_4))
    if NodeType(n) = sensor then Return F_sub
    for n_j ∈ Predecessor(n) do
        e_j = DevelopSubTree(n_j)
    for s_k ∈ Supply(r) do
        e_k = DevelopResourceSubTree(r_j, powerSource, powerLine)
    CreateGateOR(e_3, ∀e_k)
    if NodeType(n_j) = merger then
        CreateGateAND(e_2, e_j)
    else
        CreateGateOR(e_2, e_j)
    Return F_sub
procedure DevelopResourceSubTree(r, types)
    F_resSub = CreateResourceDependencyEvent(r)
    p = MappedLocation(r)
    e_1 = CreateResourceBasicEvent(r)
    e_2 = CreateLocationBasicEvent(p)
    if ((Predecessor(r) != NULL) and (ResourceType(Predecessor(r)) ∈
types)) then e_3 = CreateResourceInputEvent(r)
    CreateGateOR(F_resSub, e_1, e_2, e_3)
    if Predecessor(r) = NULL then Return F_resSub
    for r_j ∈ Predecessor(r)  do
        if ResourceType(r) ∈ types then
            DevelopResourceSubTree(r_j)
    CreateGateOR(e_3, ∀r_j)
    Return F_resSub
```

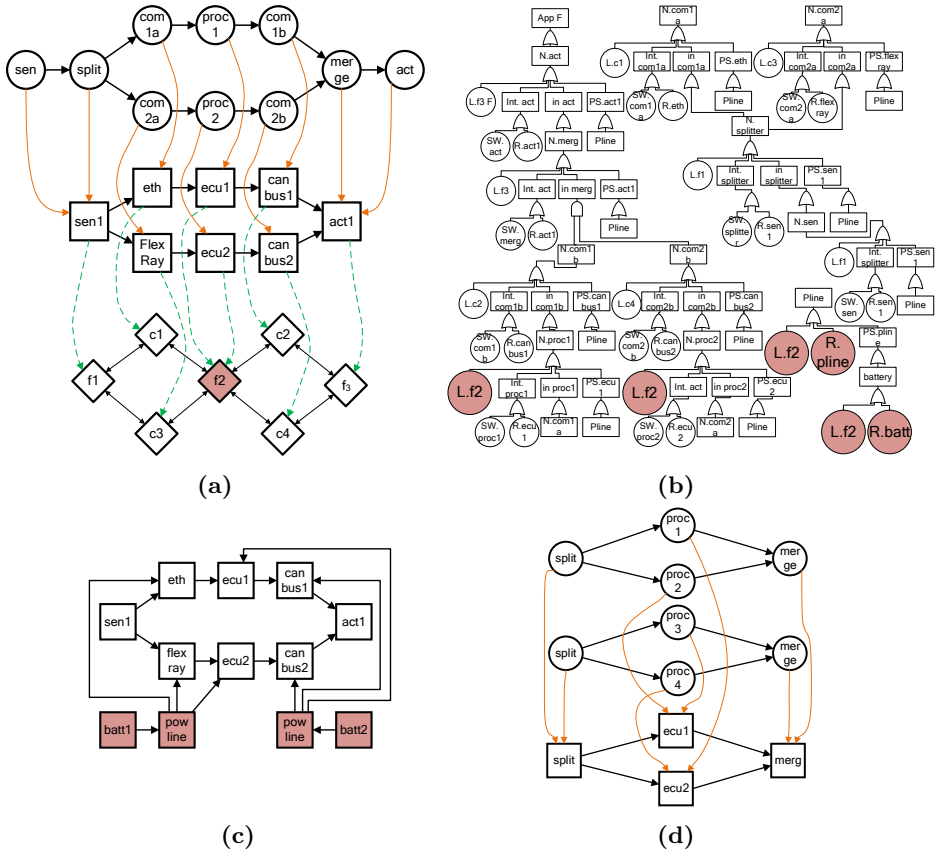

Fig. 6. Example of different scenarios and the CCF analysis results

failure of a single supply will lead to a system failure. The designer is again warned about the possible CCF, which invalidates the ASIL decomposition.

Figure 6d shows an example in which two applications have redundant elements. Since the two applications are independent from each other, it is possible to map them on the same independent resources. The ASIL decomposition assumptions will be valid, in this scenario no CCFs are present.

Figure 6b shows the generated fault tree for the first scenario, when considering a single power line and a battery to supply all the resources. The base events marked in red represent the CCF related to the placement of the redundant ECUs to the same locations and their common power supply. For this simple illustrative scenario the fault tree contains 59 events, 32 OR gates and one AND gate. In a realistic automotive system the number of nodes for each safety requirement is higher, and in combination with the high number of safety requirements the resulting graph will contain thousands of nodes. Without an automated framework, the safety engineers would have to manually create and

maintain those graphs, resulting in a more time consuming safety analysis and validation of the system, with possibilities for human errors.

6 Related Work

In [12] the authors present a method to allocate the Safety Integrity Levels (SIL) in a top-down automated process, supported by the commercial tool HiP-HOPS. The system architecture is analyzed to find which elements affect the different safety requirements and a top-down allocation of the minimum SIL values is performed. Our approach differs from theirs since we modify the architecture with model transformations in order to satisfy redundant safety requirements with new independent resources. However, the two methods can be used in combination during different phases of the design to validate each other's results.

Additional information about ASIL decomposition are given in [4] and [17], where it is made clear that introducing redundancy in safety-critical systems is a difficult task and must be taken care of by making appropriate considerations: adding an additional resource without considering its position inside the system and its dependencies is not enough for a valid decomposition.

In [6] Dynamic Fault Trees and their analysis are used to provide information about the reliability of automotive systems using the STORM tool. Since our fault trees are generated in the common Galileo format, they can be analyzed with commercial or open-source tools like STORM to obtain parameters such as the Mean Time To Failure of the system. An equivalent model that can be used for the safety analysis instead of the Fault Tree is the Reliability Block Diagram [3], but we decided to use Fault Trees to focus on the failure of the safety requirement.

In [10] the authors introduce a model for the wires used in an automotive system, optimizing the wire routing to minimize the harness expenses. This model contains more details related to these than our model, and could be used to describe the physical connections in a more refined way.

The authors in [14] describe an approach that supports the mapping of software elements to hardware resources, using the AutoFocus3 tool, in an AUTOSAR and ISO 26262 context. We currently perform the mapping step manually. A similar Design Space Exploration that considers tasks scheduling, latency, and costs is necessary to automate this process.

The work presented in [16] takes a step into the direction of a generalized functional architecture for autonomous vehicles: currently there is no standard Autonomous Driving system, but a step towards a common solution is necessary to speed up the development and validation parts, included the safety case analysis. With our work provide an environment that allows a system architect to describe generic automotive systems to compare them and decide on the most efficient solutions. It will help determine which will be the trends and most appropriate decisions for the future automotive systems. For example, the discussion between Centralized [15] versus Distributed [9] architecture design, but also more types of architectures like Domain-based [13] or more recent ideas like

Zonal [2], makes more sense when compared on a real system. They all have their pros and cons. By modeling the system and compare the same applications with different topologies it is possible to determine the efficiency of each solution.

7 Conclusions

In this work we presented a system-level analysis that validates an ASIL decomposition according to the ISO 26262 standard. We focus on the validation of the decomposition applied on a transformed system architecture, in which the designer has introduced redundancy via specific elements, hence a bottom-up method for the development of the redundant parts of the system.

Our validation is based on a CCF analysis performed on fault trees generated from the system architecture model. The model describes the system in terms of *applications*, *resources*, and *physical* layers and their mappings. The model, the fault tree generation, and the CCF analysis are implemented in Python, using the *graph-tool* library.

Our results show how a structured method to the ASIL decomposition process is necessary for a formal validation of the redundant system. We have seen that, even for a simple and artificial scenario, the generated fault trees contain a high number of events, making the CCF analysis a complex task to perform manually. By automating part of the safety analysis, the development of a safety-critical product becomes faster and less prone to human error.

Acknowledgements. The work in this paper is supported by TU/e Impuls program, a strategic cooperation between NXP Semiconductors and Eindhoven University of Technology. The authors thank all reviewers for their helpful comments and suggestions that helped improve and clarify this paper.

References

1. Boland, J.P., Proschan, F.: The Reliability of K Out of N Systems. The Annals of Probability **11**(3), 760–764 (1983)
2. Brunner, S., Roder, J., Kucera, M., Waas, T.: Automotive E/E-Architecture Enhancements by Usage of Ethernet TSN. In: 13th Workshop Intelligent Solutions in Embedded Systems. pp. 9–13 (Jun 2017). https://doi.org/10.1109/WISES.2017.7986925
3. Čepin, M.: Reliability Block Diagram, pp. 119–123. Springer London, London (2011). https://doi.org/10.1007/978-0-85729-688-7_9
4. D'Ambrosio, J.G., Debouk, R.: ASIL Decomposition: The Good, the Bad, and the Ugly. Tech. rep., SAE Technical Paper (2013)
5. Galileo User's Manual & Design Overview. https://www.cse.msu.edu/~cse870/Materials/FaultTolerant/manual-galileo.htm
6. Ghadhab, M., Junges, S., Katoen, J.P., Kuntz, M., Volk, M.: Model-based safety analysis for vehicle guidance systems. In: Tonetta, S., Schoitsch, E., Bitsch, F. (eds.) Computer Safety, Reliability, and Security. pp. 3–19. Springer International Publishing, Cham (2017)

7. IEC 61508 Edition 2.0. Principles and Use in the Management of Safety (2010)
8. ISO 26262-2011: Road vehicles - Functional safety - Part 9: ASIL-oriented and Safety-oriented Analyses (2011)
9. Jo, K., Kim, J., Kim, D., Jang, C., Sunwoo, M.: Development of Autonomous Car - Part II: A Case Study on the Implementation of an Autonomous Driving System Based on Distributed Architecture. IEEE Transactions on Industrial Electronics **62**(8), 5119–5132 (2015). https://doi.org/10.1109/tie.2015.2410258
10. Lin, C.W., Rao, L., D'Ambrosio, J., Sangiovanni-Vincentelli, A.: Electrical Architecture Optimization and Selection-Cost Minimization via Wire Routing and Wire Sizing. SAE International Journal of Passenger Cars-Electronic and Electrical Systems **7**(2014-01-0320), 502–509 (2014). https://doi.org/10.4271/2014-01-0320
11. McKelvin, Jr., M.L., Eirea, G., Pinello, C., Kanajan, S., Sangiovanni-Vincentelli, A.L.: A Formal Approach to Fault Tree Synthesis for the Analysis of Distributed Fault Tolerant Systems. In: Proceedings of the 5th ACM International Conference on Embedded Software. pp. 237–246. EMSOFT '05, ACM, New York, NY, USA (2005). https://doi.org/10.1145/1086228.1086272, http://doi.acm.org/10.1145/1086228.1086272
12. Papadopoulos, Y., Walker, M., Reiser, M.O., Weber, M., Chen, D., Törngren, M., Servat, D., Abele, A., Stappert, F., Lonn, H., et al.: Automatic Allocation of Safety Integrity Levels. In: Proceedings of the 1st Workshop on Critical Automotive Applications: Robustness & Safety. pp. 7–10. ACM (2010)
13. Reinhardt, D., Kucera, M.: Domain Controlled Architecture - A New Approach for Large Scale Software Integrated Automotive Systems. In: Pervasive and Embedded Computing and Communication Systems. vol. 13, pp. 221–226 (2013)
14. Schtz, B., Voss, S., Zverlov, S.: Automating design-space exploration: Optimal deployment of automotive sw-components in an iso26262 context. In: 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC). pp. 1–6 (June 2015). https://doi.org/10.1145/2744769.2747912
15. Sommer, S., Camek, A., Becker, K., Buckl, C., Zirkler, A., Fiege, L., Armbruster, M., Spiegelberg, G., Knoll, A.: RACE: A Centralized Platform Computer Based Architecture for Automotive Applications. In: IEEE International Electric Vehicle Conference. pp. 1–6. IEEE (2013)
16. Ulbrich, S., Reschka, A., Rieken, J., Ernst, S., Bagschik, G., Dierkes, F., Nolte, M., Maurer, M.: Towards a Functional System Architecture for Automated Vehicles. arXiv preprint arXiv:1703.08557 (2017)
17. Ward, D.D., Crozier, S.E.: The uses and abuses of ASIL decomposition in ISO 26262. In: 7th IET International Conference on System Safety, incorporating the Cyber Security Conference. pp. 1–6. IET (2012)