Towards Model-Based Failure-Management for Automotive Software

Vina Ermagan, Ingolf Krueger, Massimiliano Menarini
University of California, San Diego
{vermagan, ikrueger, menarini}@ucsd.edu

Jun-ichi Mizutani
TOYOTA InfoTechnology Center Co., Ltd.
mizutani@jp.toyota-itc.com

Kentaro Oguchi, David Weir
TOYOTA InfoTechnology Center, USA
{oguchi, dweir}@us.toyota-itc.com

Abstract

Failure management is a particular challenge problem in the automotive domain. Today’s cars host a network of 30 to 80 electronic control units (ECUs), distributed over up to five interconnected in-car networks supporting hundreds to thousands of software-defined functions. This high degree of distribution of hard- and software components is a key contributor to the difficulty of failure management in vehicle.

This paper addresses comprehensive failure management, starting from domain models for logical and deployment models of automotive software. These models capture interaction patterns as a critical part of both logical and deployment architectures, introducing failure detection and mitigation as “wrapper” services to “unmanaged services”, i.e. services without failure management. We show how these models can be embedded into an interaction-centric development process, which captures failure management information across development phases. Finally, we exploit the failure management models to verify that a particular architecture meets its requirements under the stated failure hypothesis.

1. Introduction

In the past, when the subsystems in the car were largely independent of one another, failures could be managed on a per-component basis. The shift from monolithic to highly networked, heterogeneous, interactive systems has led to a dramatic increase in both development and system complexity.

In the automotive domain, the need for fault resilience is strongly driven by passenger safety considerations. A defective deployment of an air bag, for instance, can have fatal consequences. Future X-by-wire applications (such as steer-by-wire), where safety critical functions traditionally performed by mechanical parts are accomplished by software components, will further increase the demand for failsafe at the level of software. Failure management, therefore, is a highly important topic for a mass-market product such as cars – with strong societal impact.

Effective failure management requires a thorough understanding of the application domain, including the types of failures that need to be managed, the logical and deployment architectures, the mechanisms for detecting and mitigating failures, as well as process support for the fail-safety of a given implementation. Clearly, there exist systematic approaches to failure management in general, including Failure Mode and Effects Analysis (FMEA) [1] and Fault Tree Analysis (FTA) [2]. These two techniques are very well established and applied throughout the system development process in the automotive industry. However, the increasing distribution and complexity of automotive software demand an approach that takes the cross-cutting nature of functionality and failures into account -- to complement the component-centric top-down and bottom-up approaches of FMEA and FTA, respectively.

The core contribution of this paper is an ontology for model based development of fail safe software systems and demonstration of its efficacy for modeling and verification of fail-safe automotive software. Specifically, we present (a) a taxonomy of automotive software failures, (b) interaction-based models for logical and deployment vehicle architectures, and (c) a mapping between these architectures, enabling the seamless capturing of failures across development phases. Furthermore, to explore the usefulness of the proposed models, we use them in a verification experiment. Failures are captured in this context as deviations from the expected interaction behavior.
While there is more to do to meet all the demands of [3] on a comprehensive system safety approach, these contributions constitute steps towards achieving this goal for the domain of automotive software.

The remainder of the paper is structured as follows. In Section 2 we introduce the various models that are part of our failure management ontology. In Section 3, using a simplified Central Locking System as an example, we show how to exploit this ontology for modeling failures seamlessly throughout the development process; furthermore, we demonstrate the utility of the ontology for verifying that a particular implementation meets its failure hypothesis. In Section 4 we discuss our approach in the context of related work. Section 5 contains our conclusions and outlook.

2. Models

We introduce a taxonomy for failures that facilitates capturing the required information regarding failure management in the early phases of the development process. In addition, in the spirit of Model Driven Architecture (MDA) [4], we propose a logical architecture, based on interactions, as a representation of a Platform Independent Model (PIM), and a deployment model as a representation of a Platform Specific Model (PSM). Finally, we show how the two models can be mapped together, providing the automotive engineers with a seamless approach towards fail-safe automotive software.

2.1. Failure taxonomy

A failure taxonomy is highly dependent on the application domain for which it is established [3]. In the following, we will discuss a base failure taxonomy (see Figure 1) for reactive systems in general and automotive systems in particular. We have created this ontology in a collaborative project with Toyota ITC, aiming at a domain language automotive engineers are familiar with.

Each Component in the system might fail independently of other components. A Failure is defined as a deviation from the expected (specified) behavior. In contrast an error is a design flaw (a deviation from the desired intended behavior). A system without any design error might still fail if the environment is different from the one assumed by the designer.

Each Failure leads to a number of FailureEffects that can be categorized as: Nonhazardous, Potentially Hazardous, and Hazardous. Hazard is a state in the system, which, together with the conditions in the environment, will inevitably lead to an Accident. Potentially Hazardous and Hazardous effects might lead to an Incident. An Incident is an unwanted event, which might not have any losses as a consequence. Accident is a specific type of Incident, which will cause a loss as a consequence, for instance a car accident! Safety is defined as freedom from Accidents.

![Figure 1. Failure taxonomy](image)

In the automotive domain we can observe different types of failures. We have identified three orthogonal ways of categorizing failure types. First, failures can be Software Failures or Hardware Failures. Another division is Permanent versus Temporary failures. Permanent Failures are failures that will happen every time the related function is executed in the system. For instance, a broken light bulb in a direction indicator will always fail to fulfill its function – as does a buggy software function in the system. A Temporary Failure, on the other hand, might happen every once in a while, but not every time the function is executed. Software race conditions, which might be observed as out-of-order messages on a CAN bus are examples of Temporary Failures. Failures can also be categorized as Nonoccurrence Behavior versus Unexpected Behavior of the system [5]. A system might fail by not doing what it is supposed to do. This would be a Nonoccurrence Failure. An example is when a message specified to be sent is not sent at run time. In order to build a safe system, we also need to consider the possibility of the system doing something that it is not supposed to do: an Unexpected Behavior. A duplicated CAN Bus message is a simple example of Unexpected Behavior.

Mechanisms for failure detection and mitigation establish the link between this taxonomy and the failure management domain models introduced in Section 2.2, below. First the system must be able to detect the occurrence of a failure, and then it should have a means to mitigate the faulty situation. Entities responsible for these tasks are tightly related to failures and are captured in the failure taxonomy. In the model, failures express themselves via failure effects, which are de-
ected by corresponding Fault Detectors. Fault Detectors activate Mitigators upon occurrence of a failure. Mitigators are responsible for steering the system from an unwanted state to a correct state. To this end, Mitigators have Mitigation Strategies, which are rules describing how the detected situation can be resolved. Ignoring a duplicated message, resending a lost message, activating a restricted “safe” operation mode of some functionality, and replication are examples of Mitigation Strategies. The identification of the Cause of failures can be very valuable for applying the correct Mitigation Strategy. Fault Detectors, Mitigators, Mitigation Strategies and Failure Causes are all dependent on the related failure type and can all be categorized based on this concept; in this sense, the taxonomy can be easily adapted to the concrete needs of an automotive manufacturer, and their failure management process.

While some additional work on the proposed taxonomy, such as, for example, further exploration of failure causes and their corresponding mitigation strategies, could refine it more towards the specifics of the automotive domain, the failure taxonomy described above provides a unified base model for capturing failures in the automotive domain.

An important element to apply the failure model to any system model is a specification constraining failures that happen during execution. This constraint indicates how components and connections fail in the physical model, and how services, roles, and channels fail in the logical model, along with global system constraints (such as the maximum number of concurrent failures allowed). We call this specification the failure hypothesis.

2.2. Logical model

Our approach is based on the observation that automotive failures are often cross-cutting in the sense that they originate from the interplay of different components, or that multiple components are affected, or multiple components have to coordinate to mitigate the faulty status of the system [3]. In the past, automotive engineers have successfully addressed component-failures using error codes. While this works well in a purely component-oriented environment, this approach reaches its limits when behaviors start to cross-cut multiple components. In order to address these concerns we have chosen a service oriented approach as a basis for our failure management ontology.

The main entities of the model (Figure 2) are Services, which capture the various functionalities of the system. These services are sometimes also referred to as system functions in the automotive domain. Services identify partial behaviors of the system in terms of interaction patterns. To specify interactions we use message sequence charts. Each interaction consists of Interaction Elements. An Interaction Element is either simple (an Atom) or composed of an Operator and its Operands. Messages and References (to other interaction elements) are examples of simple interaction elements. Composite interaction specifications are labeled with an operator and have a set of operands. Examples of operators are sequential and parallel composition, alternatives, and loops.[6]

The interaction-based model described so far provides the basics for specifying regular behaviors of a system in terms of composite interaction patterns.

To make the system capable of handling failures, two main additions, Detection and Mitigation, are needed. Our model adds these elements retaining the
ability to specify services that do not need fault management. This helps keeping the system specification simple whenever possible. The model defines two subtypes of Service: Unmanaged Services and Managed Services. Unmanaged Services define system behavior without considering failures. Managed Services require fault management. Following the decorator pattern [7], a Managed Service includes another service within it, which describes the interaction scenario that needs to be fail-safe. This service can be an Unmanaged or a Managed Service, giving the model the ability to create multiple layers of fault management. We also introduce: Fault Detector and Mitigator entities. A Fault Detector is responsible for monitoring the service defined inside the Managed Service and to detect eventual failures. Upon occurrence of a failure, the Fault Detector will activate the proper Mitigator, responsible for managing it. Mitigation Strategies describe what should be done when a specific type of failure happens. Mitigators include a service that specifies their behaviors.

Fault Detector detects the failures by observing their Effects. This is done by means of different Detection Strategies. For example, one Detection Strategy can be based on observing interactions; another strategy could be based on observing the internal states of relevant components.

A common mechanism for detecting failures is using time-outs. Time is captured in the model in the form of Deadlines for Interaction Elements. An Effect of a failure could be a missed Deadline.

The interaction-based model described above offers the engineer a flexible, incremental approach for creating fail-safe systems. The regular behavior of a system can be described by means of interactions within Unmanaged Services. If, later on, the need arises for adding fault management, engineers will only need to introduce Fault Detectors and Mitigators related to Unmanaged Services.

2.3. Physical model and mapping

In distributed real time systems (such as automotive systems), failures happen at both the logical and the deployment levels; in the interest of seamlessness, however, it is useful to trace these failures to the related logical entities. An error in the code is an example of logical failure, while a defective memory register is an example of deployment failure.

Figure 3 depicts a simplified physical deployment model for the automotive domain. Its main active entities are Components. Actuators, Sensors, and Controllers are examples of different component types in automotive systems. Components (following the composite pattern [7]) can be simple or compound. Components have Ports, which are entry points for Connections to other Components. One example for a connection is a Wire with electrical Signals passing over it—another example is a wireless connection.

![Figure 3. Deployment model](image)

In order to handle both physical and logical failures we need a mapping between the two layers. Figure 4 depicts how we handle this mapping in our model.

![Figure 4. Mapping model](image)

Roles are mapped to Components. Multiple Roles can be mapped to the same component. Channels are mapped to Connections. Again, multiple channels can be mapped to the same Connection. A Message is mapped to one or more Signals. This mapping supports replication of roles and channels as they might be used for mitigation strategies.

Based on this mapping, if a component/connection can fail, as a consequence, all the roles/channels mapped to that component/connection can fail as well. This establishes traceability from the logical to the deployment model.

3. From models to failure management: an automotive example

The failure domain model presented in Section 2 lends itself as the basis for a comprehensive failure management approach, consisting of means to specify failures, detectors, mitigators, and failure hypotheses seamlessly in the logical and deployment architectures. These models can be used as the basis for (1) a precise requirements notation shared by OEM and supplier, (2) integrating failure management into model-driven development processes, and (3) developing model-based tools for code generation, as well as verification and validation.

In this section we use a Central Locking System (CLS) as a case study to illustrate the efficacy of our models in developing a fail-safe automotive system.
3.1. CLS: logical model

We focus on a specific use case of CLS: unlocking the doors when an impact occurs. This important function guarantees that passengers can escape the car after an accident. We direct the reader to [8] for a more comprehensive description of CLS.

Figure 5. Unmanaged emergency unlocking

Figure 5 shows the specification of how the car should unlock during an impact. Upon impact, an Impact Sensor (IS) will send an Impact message to CONTROL, which is the command center role of the CLS. CONTROL sends an unlck message to LM, the Lock Manager, upon receipt of the Impact message. The service ends by CONTROL receiving the acknowledgement of the unlocking from LM.

Figure 6. Emergency unlocking detector

A safety property of the system requires that, whenever an impact is sensed, the doors unlock within 10 milliseconds. We create a managed service. An interaction-based detection strategy is used in UNLK-3 Detector. It monitors the interactions within the UNLK-3 as illustrated in Figure 6. UNLK3D is a new role added to the model that monitors every message passed among the roles collaborating in this service by observing the messages broadcast on the bus. Upon recognition of an Impact message from IS to CONTROL, UNLK3D starts a timer. DTC is an operator used only in Detectors. It triggers the referenced operand service if the time between the Impact message and the unlock confirmation is greater that 10ms. After detecting such a failure, one possible mitigation strategy is for UNLK3D to resend the unlock message directly to the LM. This is illustrated in the UNLK-3_Mitigator service depicted in Figure 7.

Figure 7. Emergency unlocking mitigator

3.2. CLS: deployment architecture, failure hypothesis and mapping

Figure 8 shows a specific deployment architecture for the CLS. Here we have used a CAN Bus as the connection among a number of subsystems, and wireless connections for the KeyFob subsystem.

The same diagram expresses also part of the failure hypothesis. All components except the lock manager can have hardware failures, characterized as permanent and non-occurrence based on the failure taxonomy described in section 2.1. We have chosen to consider the Lock Manager component as being reliable to simplify the example; standard replication techniques can be used to achieve this goal. The wireless connection can fail in the same way too. The failure hypothesis for the CAN Bus is a little more restrictive. It can fail only by losing one message per run. The failure hypothesis is completed by a global constraint on the number of concurrent failures allowed. In this experiment we allow only one entity per run to fail.

The mapping between elements of the logical model and of the deployment model is also captured in Figure 8. Critical for the fail-safety of our implementation of the CLS are the last three roles mentioned: IS, LM, and UNLK3D. Because, the component we mapped IS to can fail, we replicated the role, mapping it to two distinct components. In fact, our fail-safe property requires the car to always open the doors in case of an accident. To ensure this, we need to always have at least one functional sensor that can detect the impact. The replication together with the hypothesis that the CAN Bus can lose at most one message and that there is only one failure per run ensure the correct detection of every impact. FailManager plays the role of the fault detector and mitigator.

3.3. CLS: verification of fail-safety

To demonstrate how the proposed models can be leveraged to create tools that ease the creation of fail-
safe systems, we have developed an experimental verification tool. Our goal in this application is to be able to verify that the proposed architecture is “safe”, meaning it fulfills the given set of properties even if all failures admissible under the failure hypothesis happen.

In earlier work we have developed algorithms and tools for translating interaction patterns into corresponding state machines [8]. We exploit this capability to generate the model we feed into the SPIN model checker.

We use SPIN [9] to verify the stated fail-safety property. We allow a subset of the failures we have considered in the ontology, in particular: total failures of Components and the wireless Connection and loss of a single message by the CAN Bus Connection. These restrictions on the failure hypothesis can be removed with moderate effort in future releases of the tool.

The property we want to verify is that in our car model if an impact occurs, the doors get unlocked. While verifying the desired property, we can also use the SPIN-generated counterexamples to refine the failure hypothesis. This is useful when we want to determine the failure hypothesis for an already existing deployment. The concrete hypothesis thus found can then be used to refactor the architecture towards the desired failure hypothesis.

The benefits of the model checking approach we have applied to the CLS case study are plentiful. First of all, we were able to easily discover all the constraints for deployment mapping and failure hypotheses we described at the end of the previous section. Second, all this information was gathered without having to wait for the development of a full implementation, or even a feature-complete specification. This technique empowers automotive engineers to verify the fail-safety of the specified system from the early stages of the development process onward. Therefore, with the proposed approach, it is possible for OEMs to create precise end-to-end specifications of how the various components must react to failures and provide them as part of the specification provided to suppliers.

The current implementation of the verification tool is not very scalable. We plan to move the implementa-

4. Discussion and related work

The approach we have presented above leverages ideas from Model Driven Architecture (MDA)[4], specifically the use of platform independent and platform specific models, called logical and deployment architectures, respectively, here.

Our approach presents novel ideas, and leverages concepts and results from various research efforts in Fault Tolerant systems development and Software Safety. The idea of capturing failures by leveraging application specific approaches (such as our automotive failure taxonomy), to consider the problem of safety with respect to possible failure effects, and to go beyond the technical problem of fulfilling a formal specification of some sort has been studied and introduced in the software domain by Leveson in [3].

Our approach provides a concrete taxonomy for failures in the automotive domain. Moreover, we provide a concrete specification technique that can be readily embedded in company-wide system safety efforts.

The importance of automotive failure management is evident from analyzing the requirements for the Automotive Open System Architecture (AUTOSAR [10]). AUTOSAR is a consortium aiming at setting standard for automotive hardware and software architectures. In fact, the AUTOSAR requirements document [10] clearly states that AUTOSAR should provide support for redundancy (to overcome faults) and be FMEA compatible. Our approach fulfills both requirements and takes failure management even further.

In [11] Arora et al. describe how failure tolerant systems are obtained by composing intolerant systems with two types of components: detectors and correctors, which have similarities with our detectors and mitigators specifications, respectively. The paper proves that detectors and correctors are sufficient to create fail-safe and nonmasking tolerant systems, respectively. The main differences between the approach in [11] and ours is that we explicitly consider the communication infrastructure of distributed systems.

![Figure 8. CLS deployment architecture](figure8.png)
and our detectors need only to be aware of what messages are exchanged by components and not of their internal state.

Attie et al. in [12] describe how to synthesize fault tolerant programs from CTL specifications. It uses the same approach to fault tolerance as in [11] and allows generating not only the program behavior but also detectors and correctors. Their technique is subject to the state explosion problem. We mitigate this problem by allowing hierarchical service specifications and specifying detectors and mitigators at each level of the hierarchy.

5. Conclusion and Outlook

In this paper we have presented steps towards a comprehensive failure management approach for automotive software. To that end, we have developed a failure taxonomy, as well as domain models for both the logical and the deployment software architecture in vehicles. These models recognize, in particular, that (a) failure management needs to take the interactions among subsystems in the vehicle into account; (b) failures on the deployment level map to failures on the logical level, and vice versa – this is useful to interpret runtime failures in terms of the logical model and to develop deployment architectures that manage the failure types identified during design of the logical architecture.

We have embedded these models into a development process that collects and exploits failure information throughout all stages of system development.

The proposed models and process are helpful on their own: they provide guidance to automotive software engineers as to what information to collect when during the development process to enhance systemwide failure management. The end-to-end nature of our approach also has positive consequences on the interface between OEM and supplier: because the models can capture cross-cutting aspects of behavior and the corresponding failure management, interface requirements can be shared between OEM and supplier in a much more precise way than what is typical today.

In particular, however, to demonstrate further utility of the proposed model-driven failure management approach we have conducted an experiment in verifying that a given implementation meets the stated failure hypothesis; We have applied this experimental setup to the running CLS case study of this paper.

This approach can be extended to other fields where fail-safe distributed systems are required. Extensions to the failure taxonomy, the deployment layer, and the mapping model are required to support the specific failure types and deployment architectures of such systems.

As opportunity for future research we mention two ideas for exploiting the models further: we have used constraints to capture failure hypotheses, detectors and mitigators, enabling the use of standard techniques for component synthesis from logical formulae capturing the constraints at design time. Another interesting avenue for research would be to use a rule base for specifying the desired failure management strategy, and then using this rule base at runtime to execute the strategy.

6. Acknowledgements

Our work was partially supported by the UC Discovery Grant and the Industry-University Cooperative Research Program, as well as by funds from the California Institute for Telecommunications and Information Technology (Calit2). We are grateful to the anonymous reviewers for insightful comments.

7. References