SAE INTERNATIONAL®

Automatic Detection of Incomplete and Inconsistent Safety Requirements

2015-01-0268
Published 04/14/2015

Pablo Oliveira Antonino, Mario Trapp, and Ashwin Venugopal
Fraunhofer IESE

CITATION: Antonino, P., Trapp, M., and Venugopal, A., "Automatic Detection of Incomplete and Inconsistent Safety Requirements," SAE Technical Paper 2015-01-0268, 2015, doi:10.4271/2015-01-0268.

Abstract

Evidence has shown that the lack of traceability between safety requirements and both architecture and failure propagation models is a key reason for the incompleteness and inconsistency of safety requirements, and, consequently, a root cause of safety incidents. In this regard, this paper presents checks for the automatic detection of incompleteness and inconsistency of safety requirements with respect to failure propagation models and architecture. First, the notion of safety requirements completeness and consistency was decomposed into small manageable pieces called Safety Requirement Completeness and Consistency Criteria. Breaking the complex notions of completeness and consistency into finer grains was important to allow systematic and precise elaboration of the completeness and consistency checks. Next, each Safety Requirement Completeness and Consistency Criteria was formalized using Set Theory notation, which, despite being a lightweight formalism, is sufficiently accurate to represent problem-specific knowledge, and can be used as a solid basis for automation using different technology platforms. Last, in order to concretize the checks formalized with Set Theory notation, they were realized with OCL, chosen because its expressions do not have the ambiguity of natural languages and are not difficult to use in real development environments, unlike more formal specification languages such as Z. It has been observed that these checks are solid and practical enough to support safety engineers in detecting incomplete and inconsistent safety requirements, and, consequently, for improving the detection of incompleteness and inconsistency of safety requirements with respect to architecture and failure propagation models.

Introduction

Safety requirements are fundamental artifacts in the specification of safety-critical systems, since they often result from a safety analysis of the architecture, and must ultimately be addressed by elements of the architecture [1]. Because of this key role of safety requirements in safety engineering activities, it is important to assure that they meet certain quality attributes [2]. In particular, completeness and consistency of safety requirements have been widely discussed, as evidences have shown that the lack of guidance on how to specify safety requirements is one of the main reasons for their incompleteness and inconsistency, and, consequently, a root cause of safety incidents [3].

The need to detect incomplete and inconsistent safety requirements is particularly critical because if any of the traces that are supposed to exist among these elements are missing or inconsistent, there will be a lack of evidences to justify that all the potential failures of a safety-critical system were properly analyzed and mitigated.

In this regard, the work described in this paper addresses this challenge by providing means to the development staff (e.g., safety engineers, requirements engineers, architects) for automatically detecting incompleteness and inconsistency of safety requirements with respect to failure propagation models and architecture design. The automation aspect of the contribution is particularly important because both the state of the art and the state of the practice argue that any detection mechanisms for dealing with traceability concerns should be automated since, as a system grows in size and complexity, non-automated approaches are unpractical and unrealistic [4].

This paper is structured as follows: First, we present the main drivers for establishing completeness and consistency checks and provide a general overview of how the checks were designed. Next, we present a Power Sliding Door system that is used to exemplify the checks, and then present the completeness and consistency checks in detail. This is followed by a discussion of related work, and conclusions and future work are presented last.

Drivers for Safety Requirements Completeness and Consistency Checks

A set of safety engineering goals and their instantiation by different technology platforms were the main drivers for designing the safety requirements completeness and consistency checks presented in this paper. Both aspects are discussed in this section.

Safety Engineering Goals

In general, the development of safety-critical systems follows six steps [1]: *(i)* identify the system hazards; *(ii)* identify which components of the system contribute to the identified hazards; *(iii)* specify safety requirements for each component that contributes to the hazard; *(iv)* update the components according to the safety requirements; *(v)* implement the changes made at the architecture level; and *(vi)* test if the modifications performed in the system satisfy the safety requirements and detect anomalies.

Based on these generic steps, Cleland-Huang et al. [5] analyzed documents submitted to certification authorities, journals, magazines, conference proceedings, handbooks, and industrial guidelines, and derived some safety engineering specific traceability needs that should exist among the artifacts produced at each step described above. An excerpt of it is listed in Table 1.

Table 1. Safety Engineering Traceability Needs (adapted from [5]).

1	All failures described in the failure propagation models are covered by safety requirements.
2	All safety-related requirements are satisfied by elements of the architecture.
3	Determine which regulatory codes are covered by requirements.
4	Demonstrate that all safety-related design elements are realized in the code.
5	Identify parts of the code which represent standard safety mechanisms including architectural or design mechanisms such as safety interlocks, heartbeat or fault-data redundancy, to prevent a specific hazard from occurring.
6	Demonstrate that properties specifying safety-related requirements to be model checked have been model checked.
7	Determine the potential impact of changing a requirement on its associated safety-related artifacts.
8	Determine which formal models might be impacted by a change to an environmental assumption.

The safety requirements completeness and consistency checks described in this paper address items **1**, **2**, and **10** of Table 1, since they are about incompleteness of safety requirements with respect to failure propagation models and architecture design, and about inconsistencies between them.

Instantiation by Different Technology Platforms

Evidences have shown that non-automated approaches to dealing with large-scale software are unpractical and unrealistic to be considered in industrial software development environments [4][6]. Therefore, the possibility of automating completeness and consistency checks has been one of the key drivers.

In a nutshell, the approaches for supporting automated checks described in the literature can be grouped into the following categories: *(i)* formal proofs, *(ii)* model checking, *(iii)* query languages, and *(iv)* specialists computer programs. In this regard, the checks presented in this chapter have been specified such that they could be used as a primary basis for implementation with any one of the approaches listed above.

Designing Safety Requirements Completeness and Consistency Checks

The first step towards the elaboration of completeness and consistency checks was to define the meaning of completeness and consistency of safety requirements specifications: A safety requirement is complete if it is explicitly traceable to (i) the failures that motivate its existence and to (ii) the architecture elements that address the mitigation strategies described in the requirement. A safety requirement is consistent as long as there are no contradictions among safety requirements, safety-critical architecture elements, and failure propagation models. One example of consistency is when safety requirements and the safety-critical architecture elements that address them have compatible safety integrity levels.

Next, following the compositional principle proposed by Heimdahl and Leveson [7], we decomposed the safety requirements completeness and consistency notions described above into finer-grained elements: the Safety Requirement Completeness and Consistency Criteria. Breaking the complex notions of completeness and consistency into smaller pieces that are easier to manage and to analyze was important for enabling systematic and precise elaboration of the checks.

After defining the Safety Requirements Completeness and Consistency Criteria set, they were formalized using Set Theory notation [8], which, despite being a lightweight formalism, is sufficiently accurate to represent problem-specific knowledge [9] and can be used as a solid basis for automation [10]. Actually, Set Theory has been used as means for increasing transparency in the development of safety-critical systems because the notation's understandability facilitates its use by the heterogeneous members of a development team [11].

Next, in order to concretize the checks formalized with Set Theory notation, we realized them with OCL (Object Constraint Language) [12]. OCL was chosen because *(i)* our scope is model-based engineering of safety-critical systems and OCL is the Object Management Group (OMG) standard for describing constraint rules of UML- (Unified Modeling Language) and MOF- (Meta-Object Facility) based models, and *(ii)* OCL expressions do not have the ambiguity of natural languages and, unlike more formal specification languages such as Z, their adoption in real development environments is not difficult.

Running Example

We adapted the Power Sliding Door (PSD) system from [13] to exemplify the completeness and consistency checks presented in this paper. A PSD is a module-controlled system that manages the activation of a power sliding door, based on the position of the ON/OFF switch in the overhead console, in combination with other inputs such as the vehicle speed and the presence of obstacles, such as a trapped person. For this example, we adapted the function network (cf. Figure 1) from [13], which illustrates the PSD function interactions. In a nutshell, the *Computation Vehicle Speed* calculates the vehicle speed based on the input provided by the *Wheel Rotation Speed Sensors*. The vehicle speed is then sent to the *Open/Close Door Computation*, which triggers door opening requests only if the vehicle speed is below a pre-established value, which, according to [13], is 15km/h. If the vehicle is at 15km/h or less, *the Open/Close Door Computation* notifies the *Open/Close Door Signal Generator* to trigger the *Sliding Door Actuator*, which is responsible for opening and closing the sliding door.

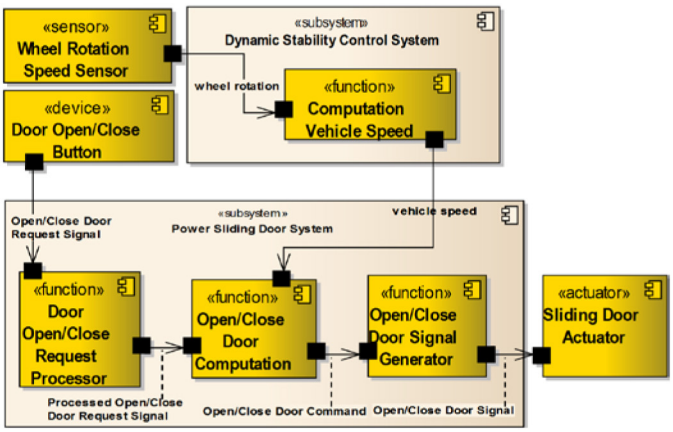

Figure 1. Power Sliding Door functions network (adapted from [13]).

We conducted a hazard analysis for the PSD system, and one of the identified hazards was: *The sliding door opens when vehicle speed is higher than 15 km/h*. We assumed that the *Open/Close Command Commission* of the *component Open/Close Door Computation* (cf. Figure 1) might lead to the occurrence of this hazard, therefore, we specified a Component Fault Tree [2] for it, as shown in Figure 2. The following combination of intermediate faults and basic event can cause the Open/Close Command Commission: The first is when the vehicle speed is higher than 15 km/h and the software that performs the speed comparison was implemented erroneously. In this case, due to the wrong implementation, it will not be detected that the vehicle speed is higher than 15 km/h. The *Logical Comparison Wrong Implemented* basic event alone characterize the second fault. The third fault can happen due to a false positive Open Door Signal, which might not be detected by a signal comparison, also because of a wrong implementation. The fourth fault is when there is a false positive of the Open Signal combined with a false positive indication that the vehicle speed is less than 15 km/h. The fifth and last fault is only due to a false positive indication that the vehicle speed is less than 15 km/h. In order to mitigate these failures as well as all the other failures of the PSD components (not shown in this paper due to lack of space), safety requirements were specified. A subset of the safety requirements described in [13] is shown in Table 2.

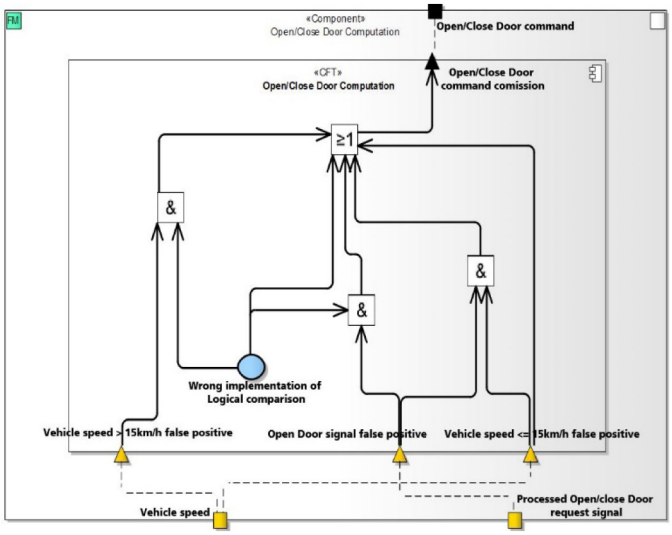

Figure 2. Open/Close Door Command Commission CFT.

Table 2. PSD safety requirements (adapted from [13]).

Safety Goal		
1.1	Not to open the door while the vehicle speed is higher than 15 km/h.	ASIL C
Functional Safety Requirements		
2.1	The door actuator will only open the door when powered by the PSDM.	ASIL C
2.2	The DSC will send the accurate vehicle speed information to the PSDM.	ASIL C
2.3	The PSDM will allow the powering of the actuator only if the vehicle speed is below 15 km/h.	ASIL C
Technical Safety Requirements		
3.1	The information about the actual vehicle speed should be updated with a cycle time of 100ms.	
3.2	The transmission of the information of the actual vehicle speed should be secured by a CRC.	
3.3	The plausibility of successive information about the actual vehicle speed should be verified by the receiver.	
3.4	The wheel rotation sensors should be diagnosed by the connected ECU.	

Safety Requirements Completeness Checks

Completeness is a quality attribute that is ensured when all the information necessary for defining or justifying a problem is found within the specification [14]. N. Levenson [15] states that a requirements specification is incomplete when the system or software behavior is not specified precisely enough, thus offering the possibility of multiple interpretations. In the safety-critical systems domain, incompleteness of safety requirements specifications as a root cause of safety-related software errors has been discussed for more than twenty years [16]. As illustrated in Figure 3, in the scope of this work, we consider a safety requirement to be complete when (i) its motivation is described in failure propagation models, which, in turn, describe the potential failures of each architectural element that might lead to a hazard occurrence concretization, and, (ii) its content describes mitigation strategies that are realized by performing modifications in the architecture, such as rearrangement of existing elements and connections and insertion of new ones.

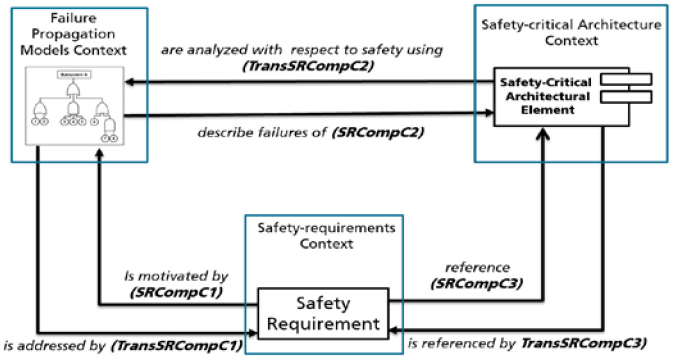

Figure 3. Bi-directional check of safety requirements completeness with respect to failure propagation models and architecture.

After partitioning this notion into finer grains, the following Safety Requirement Completeness Criteria (SRCompC) were established:

- **Safety Requirement Completeness Criterion (SRCompC1):** Every Safety Requirement (SR) describes mitigation strategies for failures that are described in at least one failure propagation model (FPM);

- **Safety Requirement Completeness Criterion 2 (SRCompC2):** Every failure propagation model (FPM) describes the failures of at least one safety-critical architecture element (SCAE); and
- **Safety Requirement Completeness Criterion 3 (SRCompC3):** Every safety requirement (SR) describes failure mitigations referencing at least one safety-critical architecture element (SCAE).

The development environment of safety-critical systems usually involves people with different educational backgrounds and professional experiences. Very often, this heterogeneity results in different languages being spoken in the development environment and different development strategies [3][14][17]. We have experienced these situations in daily development environments and have observed the consequences of this diversity; for example, even when the whole team is working on a unique specification model, it is common to find components that were inserted in the safety-critical portion of the architecture, of which the safety manager was not aware. Actually, we have seen architects make design decisions based on their own experience and ignoring the fact that, in order to make any modification in the safety-critical part of the architecture specification, there should be a safety requirement stating that a change is necessary or at least evidence that the intended modification will not compromise the overall safety of the system.

Due to this lack of communication between the different teams involved in the development of safety-critical systems, we understand that it is important to consider the transitivity property of SRCompC1, SRCompC2, and SRCompC3.

The first transitivity aspect refers to the relation between safety requirements and failure propagation models: as summarized in SRCompC1, we understand that it is important to enable engineers working in the context of safety requirements (i.e., with diagrams containing elements representing requirements) to check if every safety requirement has a failure propagation model that motivates its existence. However, we also understand that it is important to enable engineers working in the context of failure propagation models (i.e., with diagrams containing elements representing failure propagation models) to check if every failure propagation model has safety requirements associated with it (cf. Figure 3). With that, engineers working on the same model can work completely isolated and still be warned if traceability between these two artifacts is missing. This notion is summarized in the Transitive Safety Requirement Completeness Criterion 1 (TransSRCompC1):

- **Transitive Safety Requirement Completeness Criterion 1 (TransSRCompC1):** Every failure propagation model (FPM) has at least one safety requirement (SR) associated with it, describing mitigation strategies for the failures specified in it.

The second transitivity aspect refers to the relationship between failure propagation models and safety-critical architecture elements: as stated in SRCompC2, it is important to enable engineers working in the context of failure propagation models to check if every failure propagation model (FPM) describes the failures of at least one safety-critical architecture element (SCAE) specified in the architecture. However, we understand that it is also important to enable engineers working in the context of safety-critical architecture (i.e., with diagrams containing architecture elements) to check if every safety-critical architectural element (SCAE) was properly analyzed with respect to safety, even if they are working in a environment other than the one where the failure propagation models were specified (cf. Figure 3). This idea is summarized in the Transitive Safety Requirement Completeness Criterion 2 (TransSRCompC2) below:

- **Transitive Safety Requirement Completeness Criterion 2 (TransSRCompC2):** Every safety-critical architecture element (SCAE) has its potential failures described by at least one failure propagation model (FPM).

Last, as previously described in the SRCompC3, engineers working in the context of safety requirements should be able to check if every safety requirement (SR) describes failure mitigations by referencing at least one safety-critical architectural element (SCAE). However, it is also important to equip them with means to check if each of the safety-critical architecture elements (SCAE) described in the architecture specification addresses at least one safety requirement (SR) (cf. Figure 3). This notion is compiled in the Transitive Safety Requirement Completeness Criterion 3 (TransSRCompC3) described below:

- **Transitive Safety Requirement Completeness Criterion 3 (TransSRCompC3):** Every safety-critical architecture element (SCAE) addresses at least one safety requirement (SR).

In a nutshell, we understand that the completeness of safety requirements with respect to failure propagation models and architecture specification can only be ensured if the six completeness criteria described above are verified, since they offer the basis for engineers working separately to check, within their individual contexts, whether the traceability that should exist between the safety-critical systems development artifacts is met, allowing them to collaborate in the creation of the necessary evidences to justify safety. These completeness checks address the safety engineering traceability needs 1 and 2 described in Table 1: "*All failures described in the failure propagation models are covered by safety requirements"* and "*All safety-related requirements are satisfied by elements of the architecture"*. The remainder of this section details the completeness checks and provides examples with the Power Sliding Door system.

SRCompC1 and TransSRCompC1: Checks Relating Safety Requirements and Failure Propagation Models

SRCompC1: Checking Safety Requirements against Failure Propagation Models

This completeness criterion aims at enabling engineers working in the context of safety requirements (i.e., with any diagram containing elements representing safety requirements) to check if the strategies described in the safety requirements are intended for mitigating failures that are actually described in the failure propagation models (cf. Figure 3). Using Set Theory notation, this completeness criterion can be written as:

- *Let SR = {sr_1, … sr_n, sr = safety requirement of a system}*;
- *Let FPM = {fpm_1, … fpm_n, fpm = failure prop model of arch elem}*;

$$\forall\, sr \in SR: \{\exists\, fpm \in FPM: fpm \geq 1\}$$

The OCL specification of the SRCompC1 defined above with Set Theory notation is as follows:

- **Two base classes:** *SafetyRequirement* and *FailurePropagationModel*, representing the classes from which all instances of safety requirements and failure propagation models will be derived:

```
class SafetyRequirement
attributes
        name: String
        SIL: String
end

class FailurePropagationModel
attributes
        name: String
end
```

The *SafetyRequirement* base class has an attribute called SIL of the type String, indicating the safety integrity level of the safety requirement. This attribute is not used in this completeness check SRCompC1, but is considered in the checks SRCompC3 and TransSRCompC3 also detailed in this paper.

- **One association** called *isMotivatedBy*, indicating that each safety requirement describes mitigation strategies for failures that are actually described in one or more failure propagation models:

```
association isMotivatedBy between
        SafetyRequirement [1] role sr
        FailurePropagationModel [1..*] role fpm
end
```

- One OCL constraint, which is the OCL transcription of the completeness rule $\forall\, sr \in SR: \{\exists\, fpm \in FPM: fpm \geq 1\}$:

```
context SafetyRequirement
inv SRatleastOneFPM:
        SafetyRequirement.allInstances () →
        forAll(fpm →size()>0)
```

SRCompC1 Example

As SRCompC1 was designed for execution in the context of safety requirements (cf. Figure 3), the contextual starting point in the PSD example can be any diagram containing elements representing the safety requirements described in Table 2. For instance, as shown in Figure 4, the safety requirements 2.3 and 3.1 describe mitigations for failures described in the Open/Close Door Command Commission CFT depicted in Figure 2.

The OCL constraint SRCompC1 will return the Boolean value *true* only if it is verified that all safety requirements that are within the safety requirements context are explicitly associated with a CFT, as in the case of the safety requirements 2.3 and 3.1 shown in Figure 4. If at least one of the safety requirements does not match this constraint, the SRCompC1 OCL constraint will return the Boolean value *false*, indicating the existence of safety requirements that have no explicit reason for existing. With that, engineers working in the context of safety requirement have a means to ensure that every safety requirement has an explicit reason for existing.

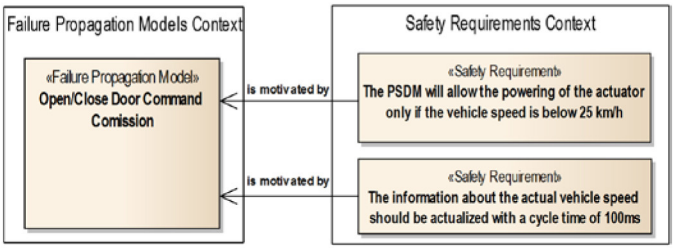

Figure 4. Safety requirements motivated by failures described in a failure propagation model.

TransSRCompC1: Checking Failure Propagation Models against Safety Requirements

This completeness criterion is a transitive property of SRCompC1 and was designed to be executed in the context of failure propagation models (cf. Figure 3). This check is important because it supports engineers in ensuring that every failure propagation model has its failures guarded by measures that are described in at least one safety requirement. Using Set Theory notation, this statement can be formally written as:

- *Let SR* = $\{sr_1, \ldots sr_n$, *sr = safety requirement of a system*$\}$;
- *Let FPM* = $\{fpm_1, \ldots fpm_n$, *fpm = failure prop model of arch elem*$\}$;

$$\forall\, fpm \in FPM: \{\exists\, sr \in SR: sr \geq 1\}$$

The OCL specification of TransSRCompC1 is as follows:

- **Two base classes:** *SafetyRequirement* and *FailurePropagationModel*, both identical to those defined for SRCompC1;
- **One association** called *isAddressedBy*, indicating that the failures of every failure propagation model are addressed by one or more safety requirements:

```
association isAddressedBy between
        FailurePropagationModel [1] role fpm
        SafetyRequirement [1..*] role sr
end
```

- **One OCL constraint**, which is the OCL translation of the completeness rule $\forall\, fpm \in FPM: \{\exists\, sr \in SR: sr \geq 1\}$:

```
context FailurePropagationModel
inv FPMatleastOneSR:
        FailurePropagationModel.allInstances() →
        forAll(sr → size()>0)
```

TransSRCompC1 Example

This completeness criterion is to be executed within the context of the failure propagation models against the safety requirements specification (cf. Figure 3). For instance, as shown in Figure 5, the Open/Close Door Command Commission CFT (foremost depicted in Figure 2) is addressed by the safety requirements 2.3 and 3.1 (foremost listed in Table 2). Briefly stated, if it is explicitly indicated that every CFT of the PSD system has its failures addressed by at least one safety requirement of Table 2, the execution of SRCompC1 OCL rule will return true; otherwise, it will return false.

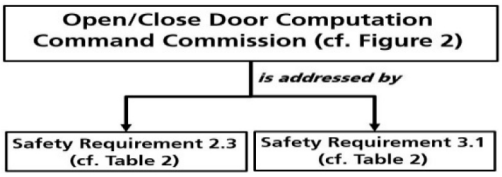

Figure 5. Open/Close Door Computation Command Commission addressed by safety requirements 2.3 and 3.1.

SRCompC2 and TransSRCompC2: Checks Relating Safety-Critical Architectural Elements and Failure Propagation Models

SRCompC2: Checking Failure Propagation Models against Safety-Critical Architectural Elements

This completeness criterion was specified to enable engineers to check, in the context of failure propagation models, whether the failures described in the failure propagation models are indeed related to safety-critical elements of the architecture (cf. Figure 3). In small projects, this issue might not be so problematic because, due to the small number of components, the engineers usually have control of the specification, and, most likely, each failure propagation model does indeed describe failures of a component that is part of the architecture specification. However, in real industrial safety-critical projects, where the specification is spread across different locations (even geographically distributed), SRCompC2 has proven its usefulness. Using Set Theory notation, this completeness criterion can be formally written as follows:

- *Let SCAE* = $\{scae_1, \ldots scae_n$, *scae = safety critical architecture element*$\}$;
- *Let FPM* = $\{fpm_1, \ldots fpm_n$, *fpm = failure prop model of architecture element*$\}$;

$$\forall\, \boldsymbol{fpm} \in \boldsymbol{FPM}: \{\exists\, \boldsymbol{scae} \in \boldsymbol{SCAE}, \boldsymbol{scae}\ \geq \boldsymbol{1}\}$$

The OCL specification of TransSRCompC1 defined above with Set Theory notation is as follows:

- **Two base classes:** *SafetyCriticalArchitecturalElement* and *FailurePropagationModel*. The latter is identical to the one defined for SRCompC1 and for TransSRCompC1:

```
class SafetyCriticalArchitecturalElement
attributes
    name: String
    SIL: String
end
```

- **One association** called *describeFailuresOf*, indicating that the failure propagation models analyze potential failures of elements that are actually part of the architecture specification:

```
association describeFailuresOf between
    FailurePropagationModel [1] role fpm
    SafetyCriticalArchitecturalElement [1..*] role scae
End
```

- **One OCL constraint**, which is the OCL translation of the completeness rule $\forall$ *fpm* $\in$ *FPM*: $\{\exists$ *scae* $\in$ *SCAE*, *scae* $\geq 1\}$:

```
context FailurePropagationModel
inv FPMatleastOneSCAE:
    FailurePropagationModel.allInstances()
    → forAll(scae → size()>0)
```

SRCompC2 Example

As SRCompC1 was designed for execution in the context of failure propagation models, in the PSD example, the contextual starting point is any diagram containing a CFT, such as the one shown in Figure 2. As illustrated in Figure 6, the Open/Close Door Command Commission CFT describes the potential failures of the Open/Close Door Computation component. However, SRCompC2 will only return the Boolean value *true* if every failure propagation model in the context of failure propagation models refers to existing elements in the architecture specification. If this constraint is not obeyed, SRCompC2 will return the Boolean value *false*, indicating to the engineers working in the context of failure propagation models that some of the failure propagation models might be describing failures that are not associated with any element of the architecture.

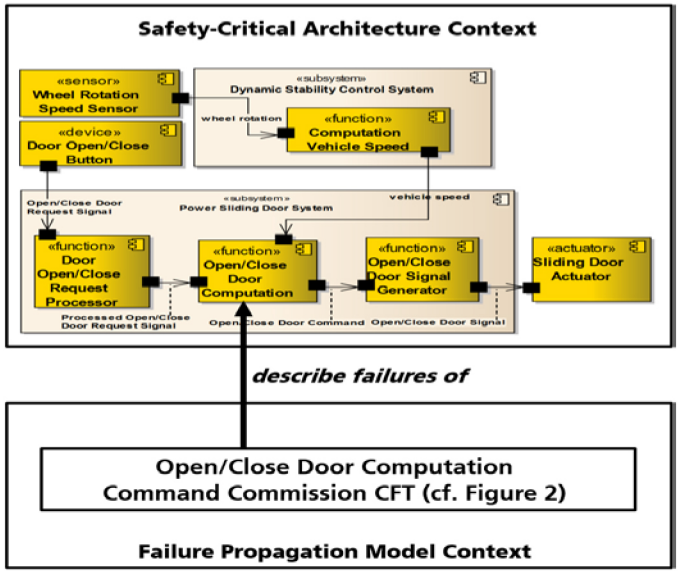

Figure 6. Failure propagation Open/Close Door Command Commission describing failures of the Open/Close Door Computation Component.

TransSRCompC2: Checking Safety-Critical Architectural Elements against Failure Propagation Models

This completeness criterion was specified to enable engineers working in the context of safety-critical architecture (cf. Figure 3) to assess whether all the architectural elements were analyzed using failure propagation models (e.g. CFT, FMEA). As already discussed, having a means to perform this check within the context of the architecture allows architects to check whether every safety-critical architecture element of the architecture was properly analyzed with respect to safety, even if they are working in an environment other than the one where the safety models were specified. Using Set Theory notation, this statement can be specified as:

- *Let SCAE* = {$scae_1$, … $scae_n$, *scae = safety critical arch elem*};
- *Let FPM* = {fpm_1, … fpm_n, *fpm = failure prop model of arch elem*};

$$\forall\ \boldsymbol{scae} \in \boldsymbol{SCAE}: \{\exists\ \boldsymbol{fpm} \in \boldsymbol{FPM}: \boldsymbol{fpm}\ \geq \boldsymbol{1}\}$$

The OCL specification of the SRCompC1 previously defined with Set Theory notation is as follows:

- **Two base classes** called *SafetyCriticalArchitecturalElement* and *FailurePropagationModel*, identical to those previously defined;
- **One association** called *areAnalyzed*, indicating that every safety-critical architectural element should be associated with at least one failure propagation model:

```
association areAnalyzed between
    SafetyCriticalArchitecturalElement [1] role scae
    FailurePropagationModel [1..*] role fpm
end
```

- **One OCL constraint**, which is the OCL translation of the completeness rule ∀ *scae* ∈ *SCAE*: {∃ *fpm* ∈ *FPM*: *fpm* ≥ 1}:

```
context SafetyCriticalArchitectureElement
    inv scaeAtLeastOnefpm:
    SafetyCriticalArchitecturalElement.allInstances() →
    forAll(fpm → size()>0)
```

TransSRCompC2 Example

For the PSD example, consider the diagram containing the functional network shown in Figure 1 as the contextual starting point. It means that the TransSRCompC2 OCL rule is to be executed within the functional network diagram. The system specified in [13] does not indicate that there are failure propagation models describing the failures that the components are subject to. However, we have specified CFTs for all of them, and one example of this can be seen in Figure 2, which depicts a CFT for the component Open/Close Door Computation. In this case, assuming that every element of the functional network described in the functional network diagram has one associated failure propagation model (cf. Figure 7), when the SRCompC1 OCL rule is executed in the context of the functional network diagram, it returns the Boolean value *true*. If at least one element of the functional network has no associated failure propagation model, the execution of the SRCompC1 OCL rule would have returned the Boolean value *false*.

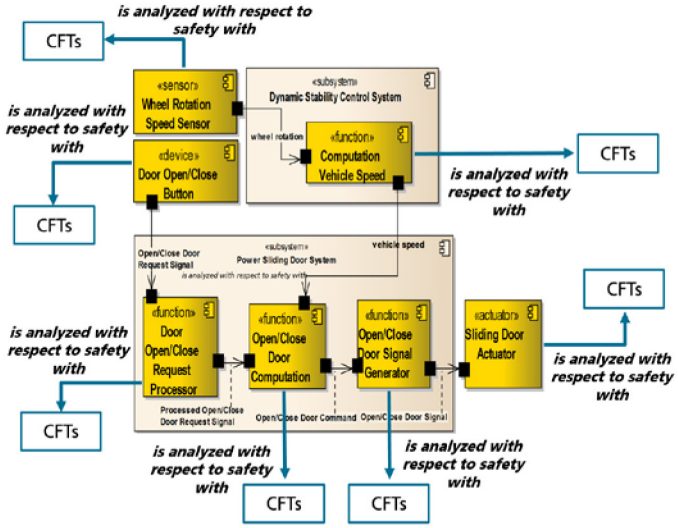

Figure 7. PSD functional network with CFTs associated with each component.

SRCompC3 and TransSRCompC3: Checks Relating Safety Requirements and Safety-Critical Architectural Elements

SRCompC3: Checking Safety Requirements against Safety-Critical Architectural Elements

Beyond ensuring that all the failures that lead to the occurrence of a hazard have associated safety requirements, it must also be guaranteed that these safety requirements are addressed by the elements described in the architecture specification. Actually, it should be explicitly indicated in each safety requirement which elements of the architecture (existing or to be included) are responsible for addressing the mitigation strategy described for it. In this regard, SRCompC3 can be formally specified using Set Theory notation as follows:

- *Let SR* = {sr_1, … sr_n, *sr = safety requirement of a system*};
- *Let SCAE* = {$scae_1$, … $scae_n$, *scae = safety critical arch element*};

$$\forall\ \boldsymbol{sr} \in \boldsymbol{SR}: \{\exists\ \boldsymbol{scae} \in \boldsymbol{SCAE}: \boldsymbol{scae}\ \geq \boldsymbol{1}\}$$

The OCL specification of SRCompC3 defined above with Set Theory notation is as follows:

- **Two base classes** called *SafetyRequirement* and *FailureCriticalArchitecturalElement*, identical to those previously defined;
- **One association** called *reference*, indicating that every safety requirement contains references to one or more safety-critical architecture elements:

```
association reference between
    SafetyRequirement [1] role sr
    SafetyCriticalArchitectureElement [1..*] role scae
end
```

- **One OCL constraint**, which is the OCL translation of the completeness rule $\forall$ *sr* $\in$ *SR*: {$\exists$ *scae* $\in$ *SCAE*: *scae* $\geq$ 1}:

context *SafetyRequirement*
inv *SRatleastOneSCAE:*
*SafetyRequirement.**allInstances()*** →
forAll(*scae* → ***size()>0)***

SRCompC3 Example

As this completeness criterion aims at checking whether the demands specified in every safety requirement are addressed by at least one safety-critical architectural element, the check has to be executed in the context of safety requirements; i.e., with any diagram containing elements representing the requirements described in Table 2.

The PSD specification described in [13] explicitly indicates which logical components address each safety requirement. For example, in Figure 8 it is shown which safety requirements foremost listed in Table 2 are addressed by which component defined in the safety-critical architecture. For the sake of simplicity, let us assume that the safety requirements context comprises only the safety requirements described in Figure 8. In this case, SRCompC3 will not return the Boolean value *true* because the safety requirements 3.3 and 3.4, marked in red, are not addressed by any safety-critical architecture element. This is an indication that it is necessary to refine the architecture to ensure that these safety requirements will be addressed.

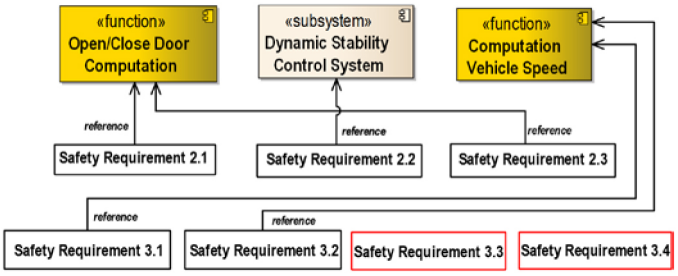

Figure 8. PSD safety requirements addressed by components of the safety-critical architecture.

TransSRCompC3: Checking Safety-Critical Architectural Elements against Safety Requirements

As mentioned above, we have seen architects make design decisions based on their own experience, ignoring the fact that, in order to make any modification in the safety-critical part of the architecture specification, there should be a safety requirement stating that a change is necessary, or at least evidence that the change to be made will not compromise the overall safety of the system. In order to address this issue, this completeness criterion was designed to be executed in the context of the architecture and aims at providing a basis for checking whether every safety-critical architectural element (SCAE) addresses at least the demands specified in one safety requirement (SR). The specification of this completeness criterion using Set Theory notation is as follows:

Let SR = {sr_1, … sr_n, *sr = safety requirement of a system*};

- *Let SCAE* = {$scae_1$, … $scae_n$, *scae = safety critical arch element*};

$$\forall\ \boldsymbol{scae} \in \boldsymbol{SCAE}: \{\exists\ \boldsymbol{sr} \in \boldsymbol{SR}: \boldsymbol{sr}\ \geq \boldsymbol{1}\}$$

The OCL specification of TransSRCompC3 is as follows:

- **Two base classes** called *SafetyRequirement* and *FailureCriticalArchitecturalElement*, identical to those previously defined;
- **One association** called *address*, indicating that every safety requirement is addressed by one or more safety-critical architectural elements:

association *address* ***between***
SafetyCriticalArchitectureElement [1] ***role*** *scae*
SafetyRequirement [1..]* ***role*** *sr*
end

- **One OCL constraint**, which is the OCL translation of the completeness rule $\forall$ *scae* $\in$ *SCAE*: {$\exists$ *sr* $\in$ *SR*: *sr* $\geq$ 1}:

context *SafetyCriticalArchitectureElement*
inv *SCAEatleastOneSR:*
*SafetyCriticalArchitectureElement.**allInstances()*** →
forAll(*sr* → ***size() > 0)***

TransSRCompC3 Example

There are certainly components in the architecture specification that are not safety-critical. However, those that are claimed to be safety-critical architectural elements should, to some extent, collaborate in addressing the demands specified in at least one safety requirement.

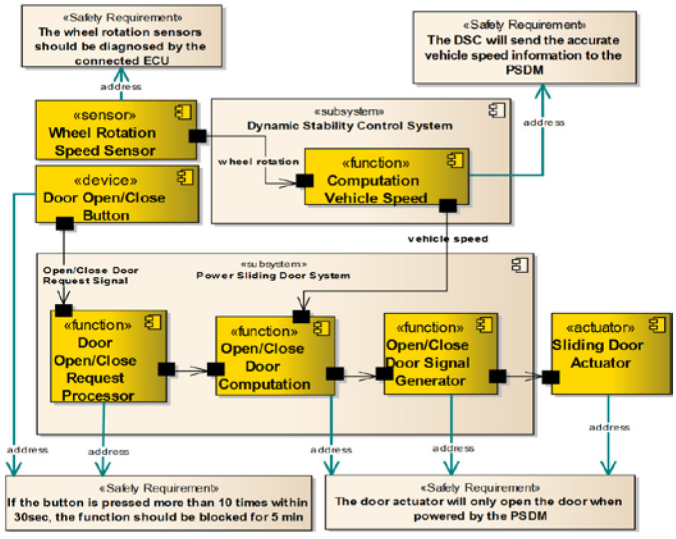

Figure 9. PSD safety-critical architecture elements addressing safety requirements.

In the architecture of the PSD system, every single component is associated with at least one safety requirement [13]. One illustration of this is shown in Figure 9. However, as previously mentioned, the opposite is not true; there are safety requirements that are not addressed by elements of the architecture. In situations like this, it is easy to see the value of performing checks between two sets of artifacts in both contexts.

Safety Requirements Consistency Checks

Consistency is a quality attribute that is ensured when there are no contradictions among development artifacts [14][16]. In the scope of this work, we understand that consistency is preserved as long as there are no contradictions among safety requirements, safety-critical architecture elements, and failure propagation models. One example of consistency is when safety requirements and the safety-critical architecture elements that address them have compatible safety integrity levels. Easterbrook and Nuseibeh [17] state that inconsistencies are mainly caused by *(i)* different development strategies, *(ii)* different languages spoken in the development environment, *(iii)* different views held by participants, *(iv)* the degree of overlap that exists in the areas of concerns of different stakeholders, and *(v)* different technical, economic and/or political objectives. In the safety-critical systems development environment, this heterogeneity is particularly augmented because of *(i)* the different educational backgrounds and different career development paths of the team members, and *(ii)* the multitude of approaches used to specify failure propagation models, architecture design, and safety requirements, which, in many cases, are absolutely unknown to the team members who are not part of the context in which the artifacts are created [18].

One trigger for the occurrence of inconsistencies among development artifacts is the modification of elements that are already referenced. In the scope of this work we stated earlier that in order to ensure completeness, it is necessary to have traces among safety requirements, failure propagation models, and safety-critical architectural elements. Once traces between these elements have been created, whenever one of them is modified, it cannot be guaranteed anymore that they are consistent. For instance, if one safety-critical architectural element (SCAE) is modified, it can no longer be guaranteed that the safety requirements (SR) and failure propagation models (FPM) related to this modified architecture element are still consistent. To address this challenge, we have specified checks to help identify which artifacts are impacted when a safety requirement or any of the elements necessary to ensure its completeness (i.e., failure propagation models and safety-critical architectural elements) is modified.

As we did for the Safety Requirements Completeness Checks described above, we also adopted the compositional principle proposed by Heimdahl and Leveson [7] for elaborating the consistency checks. In other words, we decomposed the safety requirements consistency notion stated at the beginning of this section into finer grains and called them Safety Requirement Consistency Criteria (SRConsC):

- **Safety Requirement Consistency Criterion 1 (SRConsC1):** For every updated or deleted safety requirement (UpDelSR), there are safety-critical architecture elements (SCAE), failure propagation models (FPM), and other safety requirements (SR) that are impacted;
- **Safety Requirement Consistency Criterion 2 (SRConsC2):** For every updated, deleted, or substituted safety-critical architecture element (UpDelSubSCAE), there are safety requirements (SR), failure propagation models (FPM), and other safety-critical architecture elements (SCAE) that are impacted; and
- **Safety Requirement Consistency Criterion 3 (SRConsC3):** For every updated or deleted failure propagation model (UpDelFPM), there are safety requirements (SR) and safety-critical architecture elements (SCAE) that are impacted.

Another type of inconsistency that often happens concerns safety integrity levels (SIL); more specifically, if a safety requirement and the architecture elements that address it have incompatible safety integrity levels. For example, it is indicated that the safety integrity level (or Automotive Safety Integrity Level - ASIL) of the PSD safety requirement 2.3 is “C” (cf. Table 2). This means that the architecture elements that address this safety requirement must have an equal or more stringent ASIL, which, in this case, is “C” or ASIL “D”. In this regard, we have specified a consistency check aimed at supporting the identification of SIL contradictions between safety requirements and safety-critical architectural elements, to be triggered in the context of safety requirements:

- **Safety Requirement Consistency Criterion 4 (SRConsC4):** The safety requirements are addressed by safety-critical architecture elements with an equal or more stringent safety integrity level.

Following the same transitiveness principle used in the safety requirements completeness checks, it is important to allow engineers to verify the consistency of the integrity levels in the context of the architecture, too, enabling consistency to be checked from both contexts. Therefore, we also specified a transitive check for SRConsC4:

- **Transitive Safety Requirement Consistency Criterion 4 (TransSRConsC4):** Safety-critical architecture elements address safety requirements that have an equal or less stringent safety integrity level.

Combined, these consistency checks address the safety engineering traceability need 10 described in Table 1: “*Determine the potential impact of changing a requirement on its associated safety-related artifacts”*.

Safety Requirement Consistency Criterion 1 (SRConsC1)

As stated above, we understand that a safety requirement is complete when it describes mitigation strategies for addressing failures of an architectural element, which, in turn, are systematically described in failure propagation models. This means that whenever a safety requirement is updated or deleted, its associated failure propagation models and architectural elements are necessarily impacted. In the case of hierarchical requirements specification approaches such as Safety Concept Trees [2] and the Safety Requirements Decomposition Pattern [18], there also exist dependencies among safety requirements elements in the context of safety requirements. In this case, whenever a safety requirement is modified, the safety requirements that depend on this modified one will also be impacted.

In this regard, we defined SRConsC1, which aims at identifying the safety-critical architectural elements (SCAE), failure propagation models (FPM), and other safety requirements (SR) that are impacted when a specific safety requirement is updated or deleted (UpDelSR).

In time, SRConsC1 was designed to be executed in the context of safety requirements. Using Set Theory notation, this statement can be written as:

- *Let UpDelSR = {$UpDelsr_1$, … $UpDelsr_n$: UpDelsr = updated or deleted safety requirement of a system}*;
- *Let FPM = {fpm_1, … fpm_n : fpm = failure propagation model of arch element}*;
- *Let SCAE = {$scae_1$, … $scae_n$, scae = safety critical arch element}*;
- *Let SR = {sr_1, … sr_n, sr = safety requirement of a system}*;

$$\forall \boldsymbol{UpDelsr} \in \boldsymbol{UpDelSR}: \{\exists(\boldsymbol{fpm} \in \boldsymbol{FPM}) \wedge (\boldsymbol{scae} \in \boldsymbol{SCAE}) \wedge (\boldsymbol{sr} \in \boldsymbol{SR}): \boldsymbol{fpm} \wedge \boldsymbol{scae} \geq \mathbf{1}, \boldsymbol{sr} \geq \mathbf{0}\}$$

The OCL specification of SRConsC1 defined above is as follows:

- **Three base classes** called *SafetyCriticalArchitecturalElement*, *FailurePropagationModel*, and *SafetyRequirement*:

```
class SafetyCriticalArchitecturalElement
attributes
        name: String
        SIL: Integer
        Status: String
operations
        updateSCAE()
        deleteSCAE()
        substituteSCAE()
end

class FailurePropagationModel
attributes
        name: String
        Status: String
operations
        updateFPM()
        deleteFPM()
end

class SafetyRequirement
attributes
        name: String
        SIL: Integer
        status: String
operations
        updateSR()
        deleteSR()
end
```

Observe that when these three base classes are compared with those previously specified for the safety requirements completeness checks, they were updated by the addition of an attribute of the type *Status*, which is a class attribute to which the other elements are sensitive. Actually, if any instance of a base class that contains this attribute is modified, the *Status* attribute detects that the element has been altered somehow and notifies the elements that are connected to the modified one about the changes [19]. For example, if an instance of any of these three base classes is altered, the elements related to the modified elements are notified.

The types of modifications detected by the *Status* attribute are those indicated in the *operations* specified in the base class, which, in turn, were assigned according to the different possible types of alterations these elements may be subject to. For example, observe that the base classes *FailurePropagationModel* and *SafetyRequirement* have two operations that concern update and deletion, and that the *SafetyCriticalArchitecturalElement*, has an additional operation for dealing with element substitution. The associations related to the *SafetyRequirement* base class are as follows:

- **Three associations** connecting safety requirements to failure propagation models (*isMotivatedBy*), safety-critical architectural elements (*reference*), and other safety requirements (*dependsOn*):

```
association isMotivatedBy between
        SafetyRequirement [1] role sr
        FailurePropagationModel [1..*] role fpm
end

association reference between
        SafetyRequirement [1] role sr
        SafetyCriticalArchitectureElement [1..*] role scae
end

association dependsOn between
        SafetyRequirement [1] role sr
        SafetyRequirement [0..*] role srImpacted
end
```

The OCL definition of the operations that each base class is subject to is as follows:

- **Two operations** called *updateSafetyRequirement* and *deleteSafetyRequirement*, respectively, specifying the delete and update operations that safety requirements are subject to:

```
context SafetyRequirement::updateSafetyRequirement()

    post: self.status='updated'
    post: self.scae→forAll(not(status=status@pre))
    post: self.fpm→forAll(not(status=status@pre))
    post: self.srImpacted→forAll(not(status=status@pre))

context SafetyRequirement::deleteSafetyRequirement()

    post: self.status='deleted'
    post: self.scae→forAll(not(status=status@pre))
    post: self.fpm→forAll(not(status=status@pre))
    post: self.srImpacted→forAll(not(status=status@pre))
```

These two OCL constraints are meant to be executed in the context of the safety requirement and are responsible for identifying failure propagation models (FPM), safety-critical architectural elements (SCAE), and other safety requirements (SR) that are impacted when a specific safety-critical architecture element is updated or deleted. When a safety requirement is subject to one of these two actions, we assume that the status attributes of each of the elements associated with the altered safety requirement will change (changes in the status attributes can be made manually or can be automated by a specialized program, but not by the OCL constraints, because, as OCL is only a

constraint verification language, it is supposed to check only a model's properties and execute changes on it). In this regard, when a safety requirement is updated or deleted, the OCL constraints described above check if there are failure propagation models (FPM), safety-critical architectural elements (SCAE), and other safety requirements (SR) whose status attribute is modified, as indicated in the post-conditions of the OCL constraints. If the post-conditions associated with *(i)* the failure propagation models, *(ii)* the safety-critical architecture elements, and *(iii)* the safety requirements return the Boolean value *true*, this is an indication that there are elements of each type whose status attribute is updated, which, in turn, corresponds to the set of elements that might be inconsistent because of an alteration in a safety-critical architecture element. If one of the post-conditions described in the OCL constraints returns the Boolean value *false*, this is an indication that there are no elements of that type impacted by alterations of that particular safety requirement. For instance, consider the following post-condition of the OCL constraint ***context SafetyRequirement::updateSafetyRequirement()*** described above: ***post: self.scae→forAll(not(status=status@pre))***. If it returns *false*, it means that no safety-critical architecture elements were impacted when the safety requirement was updated. We understand that the reasons for this particular example can be: (*i*) the status attribute of one of the elements referenced by the modified safety requirement was not updated due to a human or machine mistake, or *(ii)* there is a completeness violation and the safety requirement is not related to at least one safety-critical architecture element.

SRConsC1 Example

Let us consider that an engineer is working in the context of the PSD safety requirements (cf. Table 2), which, in turn, comprise safety requirements that are precisely and explicitly traced to *(i)* the function network (cf. Figure 1) and to *(ii)* the failure propagation models, such as the one shown in Figure 2. More specifically, as shown in Figure 10, let us consider that the PSD safety requirement 2.3 is explicitly linked to the component *Open/Close Door Computation* and to its failure propagation model. As can also be seen in Figure 10, each of these three elements has one *status* attribute, whose value, in a certain instant ***t0*** equals *original*. Assume that, in an instant ***t1***, a safety engineer or a requirements engineer modifies the description of the safety requirement by altering the speed at which the PSD can be opened from 15 km/h to 25 km/h, and that an external program changes the *status* attributes of the elements related to the safety requirement 2.3 from *original* to *SR updated*.

Assuming also that this model is monitored by the OCL constraints that form SRConsC1, whenever the *status* attributes change, the post- conditions described in the OCL constraints will be triggered, and, for this example, they will return the Boolean value *true*, indicating that the component *Open/Close Door Computation* and the failure propagation model *Open/Close Door Command Commission* correspond to the set of elements that might be inconsistent because of an alteration in the safety requirement 2.3.

It is important to highlight that each engineer can check in his/her work context which safety-critical architecture elements, failure propagation models, and even other safety requirements have been impacted by the change in the safety requirement (cf. Figure 10). In the example shown in Figure 10, the identification is straightforward. However, this check has proven its usefulness when the model grows in size and complexity, and when there are many engineers involved in the development.

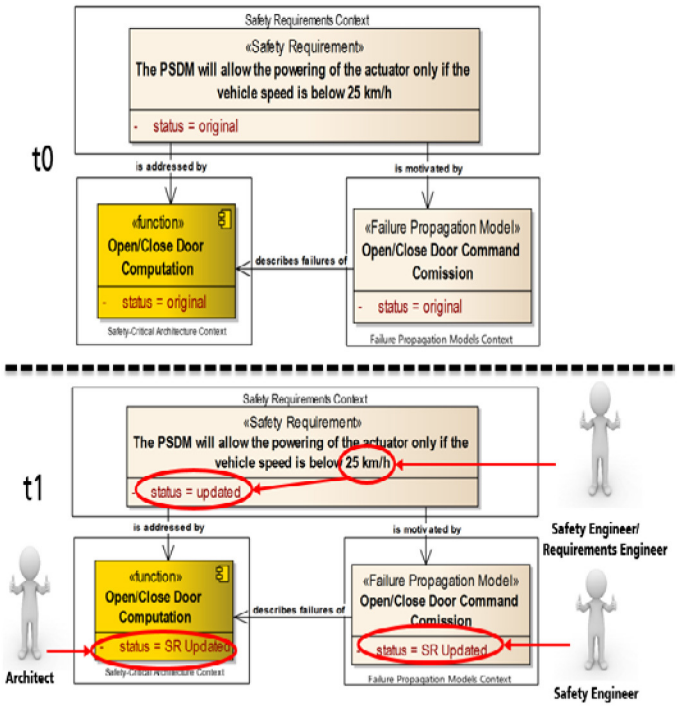

Figure 10. Detecting inconsistencies caused by modifications in safety requirements.

Due to the lack of space, the consistency checks SRConsC2, SRConsC3, SRConsC4, and TransSRConsC4 are not presented in detail in this paper. However, their specification follows the same principle of all the checks detailed in this paper.

Related Work

The works on completeness and consistency checks usually focuses on dynamic specification or static specification. Regarding the dynamic aspect, Majzik et al. [20] describe an approach for checking completeness and consistency aspects of UML statecharts diagrams. The behavioral completeness and consistency aspects they address are based on the notion that *"Completeness requires that in each basic state, for all possible events, there must be a transition defined"* and *"Consistency of the specification requires that in each state, only a single transition is triggered by a given event"*. Pap et al. [21] proposed methods and tools for automated safety analysis of UML statechart specifications aimed at identifying the correctness and completeness safety criteria proposed by Leveson. Their rules are defined in OCL and are very similar to those proposed by Majzik et al. [20]. For instance, one of the rules states that *"all states must have incoming transitions (including the initial transition)"*. The techniques presented are based on OCL expressions and graph transformations. In the same direction, Pap and Varro [22] proposed a static safety analysis technique for identifying incompleteness, non-determinism, and inconsistencies in behavioral specifications using Action Semantics descriptions. The completeness aspect addressed in the paper Static Safety Response aims at checking the existence of a response specification for every possible input sequence, including timing variations (early, late, delayed, etc. signals), and the consistency aspect is about checking the existence of

conflicting requirements and non-determinism. These are important aspects to consider, but our work is focused on the specification of static aspects of safety-critical systems.

With respect to checks on static specification, Heitmeyer et al. [23] propose techniques for performing formal analysis to automatically detect errors, such as type errors, non-determinism, missing cases, and circular definitions, in requirements specifications expressed in the SCR (Software Cost Reduction) tabular notation. Scilingo et al. [24] also proposed an approach for checking completeness, consistency, and other properties, also in SCR formal specifications, using the SPIN model checker. Aiming at preserving consistency, Van Der Straeten et al. [25] defined a formalism based on Alloi to check *horizontal and evolution consistencies*. Our approach differs from these as we have defined a clear set of artifacts that should be considered to ensure the completeness of safety requirements, whereas the related approaches are quite generic or address other consistency and completeness aspects. When it comes to model verification, the metamodel translation to Alloi proposed by Van Der Straeten et al. is less intuitive than the one we propose, and less suitable for adoption in industrial development environments. Using the combination of Set Theory and OCL proposed by us is more acceptable and preserves intuitiveness while preserving a certain degree of formality.

Conclusion

We have observed that the Safety Requirement Completeness and Consistency checks specified in this paper are solid and practical enough to support engineers in detecting incomplete and inconsistent safety requirements, and, consequently, to achieve our main goal of improving the completeness and consistency of safety requirements with respect to architecture design and failure propagation models.

As our next steps, we intend to enrich our approach by considering the relationships with other safety-specific artifacts, such as safety cases, and with validation approaches such as model-based testing.

References

1. Hatcliff J., Wassyng A., Kelly T., Comar C. and Jones P., “Certifiably safe software-dependent systems: challenges and directions,” in Future of Software Engineering (FOSE 2014), Hyderabad, India, 2014.
2. Adler, R., “Introducing Quality Attributes for a Safety Concept,” SAE Technical Paper 2013-01-0194, 2013, doi:10.4271/2013-01-0194.
3. Maeder P., Jones P. L., Zhang Y. and Cleland-Huang J., “Strategic Traceability for Safety-Critical Projects,” IEEE Software, Vol. 30, no. 3, pp. 58-68, 2013.
4. Cleland-Huang J., Goetel O., Hayes J. Huffman, Maeder P. and Zisman A., “Software traceability: trends and future directions,” in Future of Software Engineering - FOSE 2014, Hyderabad, India, 2014.
5. Cleland-Huang J., Mats H., Hayes J. Huffman, Lutz R. and Maeder P., “Trace queries for safety requirements in high assurance systems,” in 18th International conference on Requirements Engineering: foundation for software quality - REFSQ'12, Essen, Germany, 2012.
6. Antonino P. O., Keuler T., Germann N. and Cronauer B., “A Non-Invasive Approach to Trace Architecture Design, Requirements Specification, and Agile Artifacts,” in 23rd Australian Software Engineering Conference (ASWEC), Sydney, Australia, 2014.
7. Heimdahl M. P. E. and Leveson N. G., “Completeness and Consistency in Hierarchical State-Based Requirements,” IEEE Transactions on Software Engineering, Vol. 22, no. 6, 1996.
8. Halmos P. R., “Naive Set Theory”, New York, NY, USA: Springer-Verlag, 1960.
9. Guo B., Shen Y. and Zhang C., “The Hardware-Software Co-design and Co-verification of SoC for an Embedded Home Gateway,” in Fifth IEEE International Symposium on Embedded Computing, 2008. SEC '08, 2008.
10. Wang X., He F. and Liu L., “Application of Rough Set Theory to Intrusion Detection System,” in IEEE International Conference on Granular Computing, Fremont, CA, 2007.
11. Muehlhauser L., “Transparency in Safety-Critical Systems,” Machine Intelligence Research Institute, 25 08 2013. [Online]. Available: http://intelligence.org/2013/08/25/transparency-in-safety-critical-systems/. [Accessed Oct. 6, 2014].
12. Object Management Group - OMG, “Object Constraint Language,” OMG, 2014.
13. Hillenbrand M., Heinz M., Müller-Glaser K. D., Adlery N., Matheisz J., Reichman C. “An approach for rapidly adapting the demands of ISO/DIS 26262 to electric/electronic architecture modelling,” in 21st IEEE International Symposium on Rapid System Prototyping, Fairfax, VA, USA, 2010.
14. Zowghi D. and Gervasi V., “On the Interplay between Consistency, Completeness, and Correctness in Requirements Evolution,” Journal of Information and Software Technology, Vol. 46, no. 11, pp. 763-779, 2004.
15. Leveson N. G., “Safeware: System Safety and Computers”, New York, NY, USA: Addison-Wesley, 1995.
16. Glinz M. and Wieringa R., “RE@21 spotlight: Most influential papers from the requirements engineering conference,” in 21st IEEE International Requirements Engineering Conference (RE), Rio de Janeiro, Brazil, 2013.
17. Easterbrook S. and Nuseibeh B., “Managing inconsistencies in an evolving specification,” in Second IEEE International Symposium on Requirements Engineering, 1995.
18. Antonino P. O. and Trapp M., “Improving Consistency Checks between Safety Concepts and View Based. Architecture Design,” in PSAM12 - Probabilistic Safety Assessment and Management Conference, Honolulu, Hawaii, USA, 2014.
19. Stirewalt K. and Rugaber S., “Automated Invariant Maintenance Via OCL Compilation,” in Proceedings of the 8th international conference on Model Driven Engineering Languages and Systems - MoDELS'05, Montego Bay, Jamaica, 2005.
20. Pap Zs., Majzik1 I., Pataricza A. and Szegi A., Proc. IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop (DDECS'2001).
21. Pap, Zsigmond and Majzik, István and Pataricza, András, Checking General Safety Criteria on UML Statecharts. Lecture Notes in Computer Science. Computer Safety, Reliability and Security. Springer Berlin Heidelberg, 2001.

22. Pap Zsigmond, Dániel Varró: Static Safety Analysis of UML Action Semantics for Critical Systems Development. Jahrestagung der Gesellschaft für Informatik e.V. (GI), Ulm, 2004.
23. Heitmeyer Constance L., Jeffords Ralph D., and Labaw Bruce G.. "Automated consistency checking of requirements specifications". *ACM Trans. Softw. Eng. Methodol.* 5, 3 (July 1996).
24. Scilingo G., Novaira M.M, Degiovanni R., Aguirre N., "Analyzing formal requirements specifications using an off-the-shelf model checker," in CLEI 2013, pp.1,9, 7-11 Oct. 2013
25. Van Der Straeten Ragnhild, Simmonds Jocelyn, and Mens Tom. "Detecting Inconsistencies between UML Models Using Description Logic". Int'l Workshop on Description Logics DL, 2003.

Contact Information

Pablo Oliveira Antonino and Mario Trapp
Fraunhofer IESE, Embedded Systems division
Fraunhofer-Platz 1, 67663 Kaiserslautern, Germany
Pablo.Antonino@iese.fraunhofer.de
Marip.Trapp@iese.fraunhofer.de

The Engineering Meetings Board has approved this paper for publication. It has successfully completed SAE's peer review process under the supervision of the session organizer. The process requires a minimum of three (3) reviews by industry experts.

ISSN 0148-7191

http://papers.sae.org/2015-01-0268