Mälardalen University Licentiate Thesis
No.220

Enhancing the Maintainability of Safety Cases Using Safety Contracts

Omar Jaradat

November 2015

MÄLARDALEN UNIVERSITY

Department of Computer Science and Engineering
Mälardalen University
Västerås, Sweden

Copyright © Omar Jaradat, 2015
ISSN 1651-9256
ISBN 978-91-7485-238-7
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

Abstract

Safety critical systems are those systems whose failure could result in loss of life, significant property damage, or damage to the environment. These systems must be dependable and of high quality. System safety is a major property that should be adequately assured to avoid any severe outcomes. Many safety critical systems in different domains (e.g., avionics, railway, automotive, etc.) are subject to certification. The certification processes are based on an evaluation of whether the associated hazards to a system are mitigated to an acceptable level.

Safety case is a proven technique to argue about systems safety. Safety cases can provide evidential information about the safety aspect of a system by which a regulatory body can reasonably conclude that the system is acceptably safe. The development of safety cases has become common practice in many safety critical system domains. However, safety cases are costly since they need significant amount of time and efforts to produce. This cost is dramatically increased (even for already certified systems) if the changes require the safety case to be updated and submitted for re-certification. A reason for increased cost is that safety cases document highly interdependent elements (e.g., safety goals, evidence, assumptions, etc.) and seemingly-minor changes may have a major impact. Anticipating potential changes is useful since it could reveal traceable consequences that can reduce the maintenance efforts. However, considering a complete list of anticipated changes is difficult.

Safety contracts have been proposed as a means for helping to manage changes. There has been significant work that discuss how to represent and to use them, but there has been little attention on how and where to derive them. In this thesis, we focus on supporting the change impact analysis as a key factor to enhance the maintainability of safety cases. We propose an approach that shows how safety contracts can be associated with a safety case's elements to highlight them once they are impacted by changes. Moreover, we propose a safety case maintenance technique which applies sensitivity analysis in Fault Tree Analysis (FTA) to determine a system's ability to tolerate changes. The technique is twofold: (1) it supports changes prediction and prioritisation, (2) it derives safety contracts to record the information of changes with the aim to advise the engineers what to consider and check when changes actually happen. We use hypothetical and real-world systems to demonstrate our proposed approaches and technique.

Swedish Summary

Säkerhetskritiska system är system för vilka fel kan resultera i förlust av människoliv, betydande skada på egendom, eller skador på miljön. Dessa system måste vara av tillförlitliga och av hög kvalitet. Systemsäkerhet är en central egenskap som måste säkerställas för att reducera risken för allvarligare konsekvenser. Säkerhetskritiska system inom områden som avionik, järnväg och fordon är föremål för certifiering enligt certifieringsprocess bygger på en utvärdering av huruvida de associerade riskerna för ett system har reducerats till en acceptabel nivå. En sådan bevisning krävs för att tillsynsmyndigheten skall kunna intyga att ett system är tillräckligt säkert. En säkerhetsbevisning är dock kostsam, eftersom den kräver en betydande mängd tid och arbete. Denna redan höga kostnaden kan öka dramatiskt vid systemförändringar, eftersom en revidering av säkerhetsbevisningen då är nödvändig. För att kunna minska dessa underhållskostnader är det vara intressant att kunna förutsäga eventuella systemförändringar.

Att förutsäga alla möjliga systemförändringar är dock komplicerat. En enklare metod är att bestämma systemegenskapernas flexibilitet mot förändringar. Känslighetsanalys har föreslagits som ett användbart verktyg för att mäta denna flexibilitet. Utöver detta har kontrakt föreslagits som ett medel för att underlätta förändringshanteringsprocessen genom deras förmåga att fånga beroenden mellan systemkomponenter. I denna avhandling använder vi känslighetsanalys för att stödja förutsägelse av förändringar och dess prioriteringar. Vi använder dessutom säkerhetskontrakt för att fånga information om förändringar. Denna information kan vägleda ingenjörerna i vad man bör tänka på och kontrollera när förändringar sker.

“O’ Lord! Increase me in knowledge”
Holy Quran (20:114)

Acknowledgments

First and foremost, I would like to thank my supervisors, Iain Bate, Sasikumar Punnekkat and Hans Hanson. Without your continuous help and support, in the past three years, this thesis would not be possible. I would also like to thank Patrick Graydon[1], your patience, discussions and opinions are truly constructive and appreciated. Thank you all for giving me the chance to become a PhD student and for believing in me.

Many thanks to my parents for their non-stop love, warm-heartedness, motivation and support. Special thanks to my wife, your support and patience made this thesis possible. I would also like to thank my sister Arwa and my brothers Mohammad, Ahmad, Abdallah and Ali you are always there when I need you. My sons Rayyan and Ibrahim, I am sorry guys for ruining many of your weekends and school breaks, I will try not to do it again.

Next, I would like to thank the head of our division Radu for his tips and support. I thank the administrative staff, Malin Rosqvist, Carola Ryttersson, Sofia Jäderén, Susanne Fronnå, et al., for facilitating all papers work and routines. I would like to thank all researchers in Mälardalen University for the wonderful moments we have shared in lectures, meetings and fika time (coffee breaks). I would like to thank all my project mates (members of SYNOPSIS) for fruitful discussions, disputes and support. I cannot leave out my office mates and friends, Husni, Irfan, Gabriel, Anita and Filip. I want to thank the football gang which warms up the cold and lazy weekends.

Finally, I would like to thank the Swedish Foundation for Strategic Research (SSF) whose financial support made this thesis possible

Omar Jaradat
Västerås, Nov 13, 2015

[1]Patrick was my advisor since I started my PhD studies in Sep 2012 until Nov 2014

List of Publications

Papers Included in the Licentiate Thesis[2]

Paper A *An Approach to Maintaining Safety Case Evidence After A System Change*. Omar Jaradat, Patrick Graydon, Iain Bate. Proceedings of the 10th European Dependable Computing Conference (EDCC 2014), Newcastle, UK, May 2014.

Paper B *Facilitating the Maintenance of Safety Cases*. Omar Jaradat, Iain Bate, Sasikumar Punnekkat. Proceedings of the 3rd International Conference on Reliability, Safety and Hazard — Advances in Reliability, Maintenance and Safety (ICRESH-ARMS 2015), Luleå, Sweden, June 2015.

Paper C *Deriving Safety Contracts to Support Architecture Design of Safety Critical Systems*. Irfan Šljivo, Omar Jaradat, Iain Bate, Patrick Graydon. Proceedings of the 16th IEEE International Symposium on High Assurance Systems Engineering (HASE 2015), IEEE, USA, January 2015.

Paper D *Using Sensitivity Analysis to Facilitate The Maintenance of Safety Cases*. Omar Jaradat, Iain Bate, Sasikumar Punnekkat. Proceedings of the 20th International Conference on Reliable Software Technologies (Ada-Europe 2015), Madrid, Spain, June 2015.

[2]The included articles have been reformatted to comply with the licentiate layout.

Paper E *Deriving Hierarchical Safety Contracts*. Omar Jaradat and Iain Bate. Proceedings of the 21st IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2015), IEEE, Zhangjiajie, China, November 2015.

Related Papers Not Included in the Licentiate Thesis

- *Automated Verification of AADL-Specifications Using UPPAAL*. Andreas Johnsen, Kristina Lundqvist, Paul Pettersson, Omar Jaradat. Proceedings of the 14th IEEE International Symposium on High Assurance Systems Engineering (HASE 2012), October 2012.
- *Towards a Safety-oriented Process Line for Enabling Reuse in Safety Critical Systems Development and Certification*. Barbara Gallina, Irfan Sljivo, Omar Jaradat. Proceedings of the 35th Annual IEEE Software Engineering Workshop (ISOLA workshop) (SEW 2012), IEEE, October 2012.
- *The Role of Architectural Model Checking in Conducting Preliminary Safety Assessment*. Omar Jaradat, Patrick Graydon, Iain Bate. Proceedings of the 31st International System Safety Conference (ISSC 13), Boston, USA, August 2013.

Contents

I

Thesis

Chapter 1

Introduction

Safety critical systems are those systems whose failure could result in loss of life, significant property damage, or damage to the environment [1]. These systems must be dependable and of high quality. System safety is a major property that should be adequately assured to avoid severe outcomes. Medical devices, commercial aircraft, trains, nuclear plants, etc. are examples of safety critical systems. Despite the awareness of the dependability levels as to what these systems require to be acceptably safe, catastrophic accidents still occur worldwide. The Clapham Junction accident of British Rail in 1988 is an example of failures in safety critical systems that failure caused a collision between two trains, 35 people died and nearly 500 were injured, 69 of them seriously [2]. Each occurrence of harm are lessons learned that help not only to ensure that similar failures do not happen again, but they also help making the world collectively safer as time progresses. This, of course, does not mean to wait for an accident to occur in order to prevent any similar future accidents [3]. System safety is not assured by chance but rather it must be engineered and evaluated in a systematic manners that might be mandated by safety standards, best practices, experts' recommendations. Hence, many safety critical systems in different domains (e.g., avionics, railway, automotive, etc.) are subject to a certification process, which is based on an evaluation of whether the associated hazards to a system are mitigated to an acceptable level. Also, many countries and industries have authorities, inspector organisations and/or regulatory bodies that are responsible to judge whether or not safety is adequately assured.

The size and complexity of safety critical systems are considerable. Without a clear demonstration for the safety performance of a system, it is difficult for inspector organisations or system engineers themselves to build a confidence in the safety performance of the system. System engineers of some safety critical systems are required to demonstrate the safety performance of their systems through a reasoned argument that justifies why the system in question is acceptably safe (or will be so). This argument is communicated via an artefact that is known as a *safety case*. The safety case is the whole safety justification that comprises every appropriate piece of evidence to make a convincing argument to support the safety performance claims [3].

Several industries worldwide have legal obligations to produce a safety case for their operation (e.g., rail, nuclear, petrochemical and some other chemical facilities). There are even other industries that have made the creation and provision of a safety case a must (e.g., defence industry) [3]. Generally speaking, the number of safety critical systems that are subject to a certification process increases by time, where safety cases are often required for certification processes to demonstrate how a regulatory body can reasonably conclude that a system is acceptably safe from the evidence available. This usage of safety cases might make their development a common practice in many safety critical system domains.

When a safety case is not maintained, the safety case argument will remain valid for a short while only because safety critical systems are expected to operate for a long period of time and frequently subject to changes during both development and operational phases. Changes can be due to changing requirements and environmental conditions, operational experience, etc. [4].

Moreover, safety critical systems can be evolutionary as they are subject to perfective, corrective or adaptive maintenance or through technology obsolescence [5]. Changes to the system during or after development might invalidate safety evidence or argument. Evidence might no longer support the developers' claims because it reflects old development artefacts or old assumptions about operation or the operating environment. Also, existing safety claims might be nonsense, no longer reflect operational intent, or be contradicted by new data. Eventually, the real system will have diverged so far from that represented by the safety case argument and the latter is no longer valid or useful [3]. Hence, it is almost inevitable that the safety case will require updating throughout the operational lifetime of the system. An additional key reason to maintain safety cases is that any change that might compromise system safety involves repeating the certification process (i.e., re-certification) and repeating the certification process necessitates an updated and valid safety case that considers the changes. For example, the UK Ministry of Defence Ship Safety Management System Handbook JSP 430 requires that *"the safety case will be updated*

... to reflect changes in the design and/or operational usage which impact on safety, or to address newly identified hazards. The safety case will be a management tool for controlling safety through life including design and operation role changes" [6, 7]. Similarly, the UK Health and Safety Executive (HSE) — Railway safety case regulations 1994 — states in regulation 6(1) that *"a safety case to be revised whenever, appropriate that is whenever any of its contents would otherwise become inaccurate or incomplete"* [8, 7].

A safety case, therefore, is not built for one use, but rather it is built as a living document that should always be maintained to justify the safety status of the associated system and evolves as this system evolves. In addition to its role in justifying system safety, a safety case should also identify and manage the impact of changes. For example, in the Clapham Junction railway case, the cause of the accident was a signal failure caused by an improper termination of old wires after installing a new wiring during maintenance. British Rail staff did not fully understand the importance of wire checks after the change. There was no safety case built for the signaling system [9]. Possibly, the existence of a safety case that describes the importance of wiring verification and how it should be done could have helped avoiding the accident.

One of the biggest challenges that affects safety case revision and maintenance is that a safety case documents a complex reality. A safety case comprises a complex web of interdependent elements, such as, safety goals, evidence, argument, and assumptions about operating context. These elements are highly interdependent and thus seemingly minor changes may have a major impact on the contents and structure of the safety argument. As such, a single change to a safety case may necessitate many other consequential changes — creating a ripple effect [5].

Any improper maintenance in a safety argument might negatively impact the system safety performance conveyed by the safety case. The improper maintenance might (1) preclude the safety case to make that performance clear to readers, or (2) change the status of that argument from sound to unsound by changing the structure of that argument, changing the truth of its premises, or both. Hence, a step to assess the impact of this change on the safety argument is crucial and highly needed prior to updating a safety argument after a system change. Despite clear recommendations to adequately maintain and review safety cases by safety standards, such as those quoted earlier, existing standards offer little advice on how such operations can be carried out [5].

The concept of contract has been around for few decades in the system development domain. There have been significant works that discuss how to represent and to use contracts [10, 11, 12]. For example, researchers have used assume-guarantee contracts to propose techniques to lower the cost of developing software for safety critical systems. Moreover, contracts have been exploited as a means for helping to manage system changes in the system domain or in its corresponding safety case [13, 14, 15]. However, using contracts as a way of managing change is not a new notion since it has been discussed in some works [16, 13], but deriving the contracts and their contents have received little support yet [4].

Sensitivity analysis helps the experts to define the uncertainties involved with a particular system change so that those experts can judge the potential change based on how reliable the consequences are. The use sensitivity analysis to define the problematic parts of a system with respect to changes. More specifically, we exploit the Fault Tree Analyses (FTAs), which developers often perform as part of safety analysis phase, and apply the sensitivity analysis to those FTAs in order to identify the sensitive parts in them. We define a sensitive part as an event whose minimum changes have the maximal effect on the FTA, where effect means exceeding reliability targets due to a change [4].

In this thesis, we combine sensitivity analysis together with the concept of contracts to facilitate the accommodation of system changes in safety cases to ultimately enhance the maintainability of safety cases. Our work focuses on:

1. how and where to derive safety contracts and their contents,
2. using the derived contracts to support the decision as to whether or not apply changes, and
3. using the derived contracts to guide developers to the parts in the safety case that might be affected after applying a change.

1.1 Thesis Outline

The thesis report is organized into two main parts. **Part I** includes six chapters. Chapter 1 has provided an introduction to the thesis where an overview of the research problem, motivation and the thesis contribution were presented. In Chapter 2, we present background information about safety critical systems, Fault Tree Analysis (FTA), safety cases and arguments, safety contracts, and sensitivity analysis. In Chapter 3, we describe the research problems and derive the research goals. We also describe the overall methodology that is adopted to run the research. In Chapter 4, we present the contributions of the research and reflect on the research goals. In Chapter 3.3, we present the related work. We draw the conclusion and describe the possible future work in Chapter 5. **Part II** contains the research papers included in the thesis, as follows:

Paper A (Chapter 6): An Approach to Maintaining Safety Case Evidence After A System Change, Omar Jaradat, Patrick Graydon, Iain Bate.

Abstract: *"Developers of some safety critical systems construct a safety case. Developers changing a system during development or after release must analyse the change's impact on the safety case. Evidence might be invalidated by changes to the system design, operation, or environmental context. Assumptions valid in one context might be invalid elsewhere. The impact of change might not be obvious. This paper proposes a method to facilitate safety case maintenance by highlighting the impact of changes."* [17]

Status: Published in Proceedings of the 10th European Dependable Computing Conference, EDCC 2014.

My contribution: I was the main contributor of the work under supervision of the co-authors. My contributions include proposing a new approach to facilitating safety case change impact analysis.

Paper B (Chapter 7): Facilitating the Maintenance of Safety Cases, Omar Jaradat, Iain Bate, Sasikumar Punnekkat.

Abstract: *"Developers of some safety critical systems construct a safety case comprising both safety evidence, and a safety argument explaining that evidence. Safety cases are costly to produce, maintain and manage. Modularity has been introduced as a key to enable the reusability within safety cases and thus reduces their costs. The Industrial Avionics Working Group (IAWG) has proposed Modular Safety Cases as a means of containing the cost of change by dividing the safety case into a set of argument modules. IAWG's Modular Software Safety Case (MSSC) process facilitates handling system changes as a series of relatively small increments rather than occasional major updates. However, the process does not provide detailed guidelines or a clear example of how to handle the impact of these changes in the safety case. In this paper, we apply the main steps of MSSC process to a real safety critical system from industry. We show how the process can be aligned to ISO 26262 obligations for decomposing safety requirements. As part of this, we propose extensions to*

MSSC process for identifying the potential consequences of a system change (i.e., impact analysis), thus facilitating the maintenance of a safety case." [18]

Status: Published in Proceedings of the 3rd International Conference on Reliability, Safety and Hazard — Advances in Reliability, Maintenance and Safety, ICRESH-ARMS 2015.

My contribution: I was the main contributor of the work under supervision of the co-authors. My contributions include showing how to apply the MSSC (Modular Software Safety Case) process to a real safety critical system to show how system engineers can identify the elements in a safety argument that might be impacted by a change.

Paper C (Chapter 8): Deriving Safety Contracts to Support Architecture Design of Safety Critical Systems, Irfan Šljivo, Omar Jaradat, Iain Bate, Patrick Graydon.

Abstract: *"The use of contracts to enhance the maintainability of safety-critical systems has received a significant amount of research effort in recent years. However some key issues have been identified: the difficulty in dealing with the wide range of properties of systems and deriving contracts to capture those properties; and the challenge of dealing with the inevitable incompleteness of the contracts. In this paper, we explore how the derivation of contracts can be performed based on the results of failure analysis. We use the concept of safety kernels to alleviate the issues. Firstly the safety kernel means that the properties of the system that we may wish to manage can be dealt with at a more abstract level, reducing the challenges of representation and completeness of the "safety" contracts. Secondly the set of safety contracts is reduced so it is possible to reason about their satisfaction in a more rigorous manner."* [19]

Status: Published in Proceedings of the 16th IEEE International Symposium on High Assurance Systems Engineering, HASE 2015.

Contributions: Irfan Šljivo, Iain Bate and I are the main contributors. Patrick Graydon reviewed the paper and provided comments for improvement at the paper. My contributions include building the safety argument before and after introducing a change. Also, associating the derived safety contracts for parts of the system design with the corresponding argument fragments as a means to establish a traceability mechanism between the system and its safety case.

Paper D (Chapter 9): Using Sensitivity Analysis to Facilitate The Maintenance of Safety Cases, Omar Jaradat, Iain Bate, Sasikumar Punnekkat.

Abstract: *"A safety case contains safety arguments together with supporting evidence that together should demonstrate that a system is acceptably safe. System changes pose a challenge to the soundness and cogency of the safety case argument. Maintaining safety arguments is a painstaking process because it requires performing a change impact analysis through interdependent elements. Changes are often performed years after the deployment of a system making it harder for safety case developers to know which parts of the argument are affected. Contracts have been proposed as a means for helping to manage changes. There has been significant work that discusses how to represent and to use them but there has been little on how to derive them. In this paper, we propose a sensitivity analysis approach to derive contracts from Fault Tree Analyses and use them to trace changes in the safety argument, thus facilitating easier maintenance of the safety argument."* [4]

Status: Published in Proceedings of the 20th International Conference on Reliable Software Technologies, Ada-Europe 2015.

My contribution: I was the main contributor of the work under supervision of the co-authors. My contributions include combining the results of sensitivity analysis together with the concept of contracts to identify the sensitive parts of a system and highlight these parts to help the experts to make an educated decision as to whether or not apply changes.

Paper E (Chapter 10): Deriving Hierarchical Safety Contracts, Omar Jaradat, Iain Bate, Sasikumar Punnekkat.

Abstract: *"Safety cases are costly since they need significant amount of time and efforts to produce. This cost can be dramatically increased (even for already certified systems) due to system changes as they require maintaining the safety case before it can be submitted for certification. Anticipating potential changes is useful since it reveals traceable consequences that will eventually reduce the maintenance efforts. However, considering a complete list of anticipated changes is difficult. What can be easier though is to determine the flexibility of system components to changes. Using sensitivity analysis is useful* *to measure the flexibility of the different system properties to changes. Furthermore, contracts have been proposed as a means for facilitating the change management process due to their ability to record the dependencies among system's components. In this paper, we extend a technique that uses a sensitivity analysis to derive safety contracts from Fault Tree Analyses and uses these contracts to trace changes in the safety argument. The extension aims to enabling the derivation of hierarchical and correlated safety contracts. We motivate the extension through an illustrative example within which we identify limitations of the technique and discuss potential solutions to these limitations."*

Status: Accepted for publication in Proceedings of the 21st IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2015.

Main contribution: I was the main contributor of the work under Bate's supervision. My contribution comprises (1) identifying possible limitations for the proposed technique in Paper D and (2) suggesting an extension to the technique to resolve the identified limitations.

Chapter 2

Background

In this chapter, we provide background and overview of the most prominent terms that appear frequently in this thesis.

2.1 Safety Critical Systems

The word 'safety' means: *"The condition of being protected from or unlikely to cause danger, risk, or injury"* [20]. *Safety critical "is a term applied to a condition, event, operation, process or item that is essential to safe system operation or use, e.g., safety critical function, safety critical path, and safety critical component"* [21]. ***Safety critical systems*** *are those systems whose failure might endanger human life, lead to substantial economic loss, or cause extensive environmental damage* [1]. That is, the operation of safety critical systems should be safe and, ideally, never cause severe consequences. However, developing absolutely safe system is unattainable even if the project has an open budget. This is because severe consequences are typically linked to system faults and we cannot be 100% certain that a system is fault free. However, this shall not discourage the efforts that aim at assuring systems' safety.

The key to assuring safety is to eliminate hazards or to ensure that the consequences of these hazards are minimal. The word ***hazard*** in English is defined as: *"a potential source of danger"* [20]. In the context of safety critical systems, there are different suggestions to explain what the word *hazard* means. Some definitions suggest that a hazard is simply a system state that could lead to accidents. For example, Knight [22] indicates that the word *hazard* is an abbreviation of ***hazard state*** and it means: *"a system state that could lead to an unplanned loss of life, loss of property, release of energy, or release of dangerous materials"* [22]. Some other definitions suggest to consider the potential environmental conditions in the definition to clarify the relationship between hazards and accidents. For example, Leveson [23] define *hazard* as: *"a state or condition of a system that, combined with certain environmental conditions, could lead to accidents"*. Anyway, all definitions agree that a hazard is a system state in which an accident might occur. An ***accident*** can be defined as: *"An unplanned event or sequence of events which results in human death or injury, damage to property, or to the environment"* [24]. The measure of the probability that a system will cause an accident is referred to as ***risk***. The risk is assessed by considering the probability that someone/something will be harmed if exposed to a hazard (also know as ***exposure***) and hazard and the severity of that hazard.

Hazards are caused by malfunctioning behaviours (i.e., failures) [25]. A ***Failure*** is defined as: *"an event that occurs when the delivered service deviates from correct service"* [26]. Failures are caused by errors. An ***error*** is defined as: *a part of the total state of the system that may lead to its subsequent service failure* [26]. Finally, errors are caused by faults. A ***fault*** is defined as: *an adjudged or hypothesized cause of an error* [26].

Figure 2.1 illustrates some of the previous system safety concepts and how they might relate to each other. The figure uses a scenario from an adaptive cruise control system[1] to give examples of these concepts.

Any process or activity that aims at assuring or improving systems' safety should identify and eliminate potential hazards of those systems. If the elimination is not possible then they should be mitigated to an acceptable level. *Preliminary Hazard Analysis (PHA)* is used to can be done with only a description of the system's concept and functions. That is, PHA is typically used in the early stages of the system's lifecycle where no enough design detail is available. PHA accomplishes four main tasks as follows [27]:

- Identify system hazards
- Translate system hazards into high-level system safety design constraints
- Assess hazards if required to do so
- Establish the hazard log

[1]Adaptive Cruise Control (ACC) system is an optional system for road vehicles that automatically adjusts the vehicle speed to maintain a safe distance from vehicles ahead.

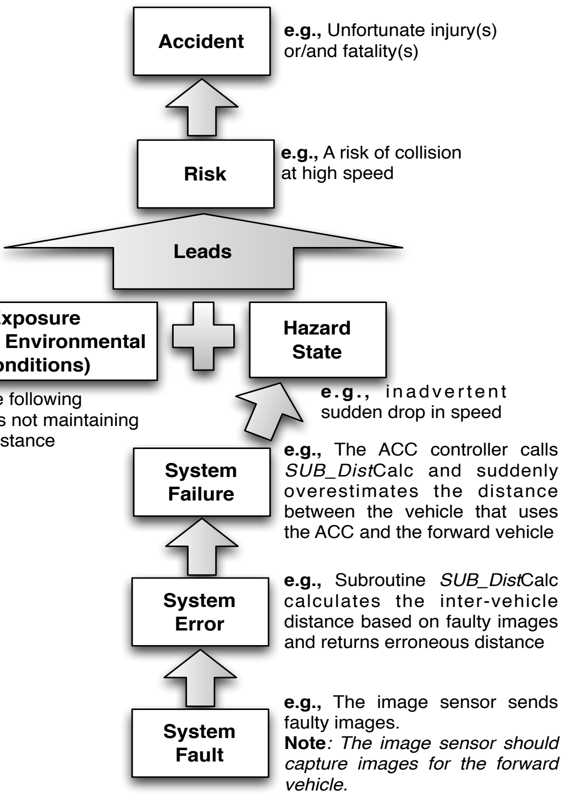

Accident
e.g., Unfortunate injury(s) or/and fatality(s)
Risk
e.g., A risk of collision at high speed
Leads
Exposure (Certain Environmental Conditions)
e.g., The following vehicle is not maintaining a safe distance
Hazard State
e.g., inadvertent sudden drop in speed
System Failure
e.g., The ACC controller calls SUB_DistCalc and suddenly overestimates the distance between the vehicle that uses the ACC and the forward vehicle
System Error
e.g., Subroutine SUB_DistCalc calculates the inter-vehicle distance based on faulty images and returns erroneous distance
System Fault
e.g., The image sensor sends faulty images.
Note: The image sensor should capture images for the forward vehicle.

Figure 2.1: Overview of some basic system safety concepts

Safety functions (also know as *Safety Barriers*) shall be identified, implemented and verified to achieve or maintain the safe state of a system (with respect to the identified hazards). These functions can be safeguards, countermeasures, or protection layers, etc. (e.g., fire and gas detection system, pressure relief system, emergency shutdown system, etc.). The reliability of such functions are crucial to achieve safety. Reliability here means *"the ability of the item to perform a required function, under given environmental and operational conditions and for a stated period of time"* [28]. However, safety should not be confused with reliability. A reliable system can be unsafe and vice versa. The software of a system may still behave in such a way that the resultant system behavior leads to an accident. The Lufthansa flight 2904 accident can be an example of how a reliable system may be unsafe. The plane was landing at Warsaw airport in Poland when the computer-controlled braking system did not work. While landing, the braking system did not recognize that the plane touched the ground and assumed that the aircraft was still airborne. A safety feature on the aircraft had stopped the deployment of the reverse thrust system, which slows down the aircraft. The plane ran off the end of the runway, hit an earth bank, and caught fire [24]. The investigations revealed that the braking system software was reliable and had operated according to its specification, but this did not lead to a safe system [24].

The acceptable safety levels of a system should be defined and all identified risks should be eliminated or reduced to these levels. There are several well-described quantitative and qualitative measures for safety functions in the literature. The work in this thesis, however, depends on failure probabilities derived from quantitative FTA as quantitative measures.

2.2 Fault Tree Analysis (FTA)

In 1962, Bell Telephone Laboratories introduced the fault tree technique as a means to evaluate safety in the launching system of the intercontinental *Minuteman* missile [29]. The Boeing Company improved the technique and introduced computer programs for both qualitative and quantitative fault tree analysis. Today FTA is the most commonly used technique for safety and reliability studies.

FTA is a failure analysis method which focuses on one particular undesired event and provides a method for determining causes of this event [30]. In other words, FTA is used to specify the occurrence of critical states (from a safety or reliability standpoint). These states might be associated with component hardware failures, human errors, software errors, or any other pertinent events. FTA helps safety engineers to identify plausible causes (i.e., faults) of undesired events [31].

A fault tree illustrates the logical interrelationships of the system's components (***Basic Events***) that lead to the undesired event or the system's state (***Top Event***) [31, 29]. These logical interrelationships are called ***Logical Gates***. Figure 2.2 shows the most commonly used FTA symbols.

Moreover, FTA is used as a method to achieve Probabilistic Safety Analysis (PSA). More specifically, it is used to quantify system failure probability. Quantitative FT evaluation techniques produce three types of results: (1)

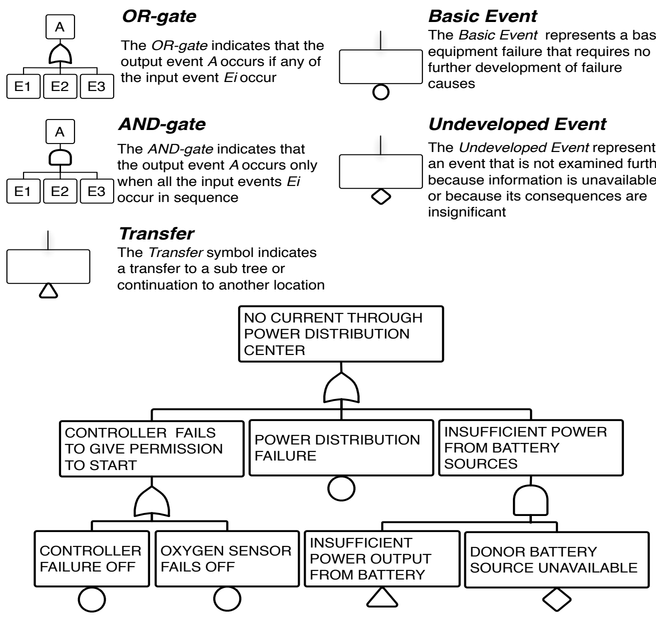

OR-gate
The OR-gate indicates that the output event A occurs if any of the input event Ei occur
A
E1
E2
E3
AND-gate
The AND-gate indicates that the output event A occurs only when all the input events Ei occur in sequence
Transfer
The Transfer symbol indicates a transfer to a sub tree or continuation to another location
Basic Event
The Basic Event represents a basic equipment failure that requires no further development of failure causes
Undeveloped Event
The Undeveloped Event represents an event that is not examined further because information is unavailable or because its consequences are insignificant
NO CURRENT THROUGH POWER DISTRIBUTION CENTER
CONTROLLER FAILS TO GIVE PERMISSION TO START
POWER DISTRIBUTION FAILURE
INSUFFICIENT POWER FROM BATTERY SOURCES
CONTROLLER FAILURE OFF
OXYGEN SENSOR FAILS OFF
INSUFFICIENT POWER OUTPUT FROM BATTERY
DONOR BATTERY SOURCE UNAVAILABLE

Figure 2.2: The principal FTA symbols and an instantiation [29, 32]

numerical probabilities, (2) quantitative importance, and (3) sensitivity evaluations [30]. In this thesis, we exploit the results obtained by sensitivity evaluations to measure how sensitive a system design is to a particular aspect of individual event. More details about our sensitivity analysis is found in Section 2.3.

2.3 Sensitivity Analysis

Sensitivity analysis can be defined as: *"The study of how uncertainty in the output of a model (numerical or otherwise) can be apportioned to different sources of uncertainty in the model input"* [33]. The analysis helps to establish reasonably acceptable confidence in the model by studying the uncertainties that are often associated with variables in models. Many variables in system analysis or design models represent quantities that are very difficult, or even impossible to measure to a great deal of accuracy. In practice, system, developers are usually uncertain about variables in the different system models and they estimate those variables. Sensitivity analysis allows system developers to determine what level of accuracy is necessary for a parameter (variable) to make the model sufficiently useful and valid [34].

There are different purposes for using sensitivity analysis. The analysis can be used to provide insight into the robustness of model results when making decisions [35]. Also, the analysis can be used to enhancing communication from modelers to decision makers, for example, by making recommendations more credible, understandable, compelling or persuasive [36]. In safety domains, sensitivity analysis can be used in risk analysis models to determine the most significant exposure or risk factors so to speak, and thus, it can support the prioritisation of the risk mitigation. Sensitivity analysis methods can be classified in different ways such as mathematical, graphical, statistical, etc. In this paper we use the sensitivity analysis to identify the safety argument parts (i.e., sensitive parts) that might require unneeded painstaking work to update with respect to the benefit of a given change. The results of the analysis should be presented in the safety argument so that it is always available up front to get developers' attention.

In this thesis, we apply the sensitivity analysis on FTAs to measure the sensitivity of outcome A (e.g., a safety requirement being true) to a change in a parameter B (e.g., the failure probability in a component). The sensitivity is defined as $\Delta B/B$, where ΔB is the smallest change in B that changes A (e.g., the smallest increase in failure probability that makes safety requirement A false). The failure probability values that are attached to FTA's events are considered input parameters to the sensitivity analysis. A sensitive part of a FTA is defined as one or multiple FTA events whose minimum changes (i.e., the smallest increase in its failure probability due to a system change) have the maximal effect on the FTA, where effect means exceeding failure probabilities (reliability targets) to inadmissible levels. A sensitive event is an event whose failure probability value can significantly influence the validity of the FTA once it increases. In this this, system components whose failure rates correspond to FTA's events whose likelihoods are sensitive are referred to as sensitive components. Hence, changes to a sensitive component cause a great impact to system design [4].

We use the sensitivity analysis as a method to determine the range of failure probability parameter for each event. Hence, our work assumes the existence of a probabilistic FTA where each event in the tree is specified by an actual (i.e., current) failure probability $FP_{Actual|event(x)}$. In addition, our work assumes the existence of the required failure probability for the top event

$FP_{Required(Topevent)}$, where the FTA is considered unreliable if: $FP_{Actual(Topevent)} > FP_{Required(Topevent)}$.

2.4 Safety Case and Safety Argument

2.4.1 Safety Case

In 1965, section 14 of the UK Nuclear Installations Act states:

"Without prejudice to any other requirements of the conditions attached to this license the licensee shall make and implement adequate arrangements for the production and assessment of safety cases consisting of documentation to justify safety during the design, construction, manufacture, commissioning, operation and decommissioning phases of the installation." [37]

Hence, the notion of building safety cases to justify safety is not new and it has been around for almost fifty years. In 1989, the British chemical industry requested from nuclear sites (according to the Control of the Industrial Major Accident Hazards (CIMAH) regulations) to generate a written report that should contain (1) facts about the site, and (2) reasoned arguments about the hazards and risks from the site [38]. This report was also known as a safety case. The objective of the report was to demonstrate to the UK Health and Safety Executive (HSE) that the site is satisfactory by listing the major hazards and risks and shows how they are adequately mitigated.

The development of the safety case as a means of demonstrating acceptable risk began in the nuclear industry but the application of this means was uncommon in other industries. For example, in the Clapham junction accident in Chapter 1, there was no safety case and although the transport system was allegedly mature, regulated and safe, British Rail could not demonstrate why their system was acceptably safe to operate [2]. From 1990s onwards the development of safety cases spread across many other major hazard industries, such as the railways, offshore oil, gas facilities, etc. [39].

2.4.2 Safety Case Definition

It is worth mentioning that in addition to the term 'safety case', there are different other terms such as '*Assurance Case*' and '*Safety Assurance Case*' that are, sometimes, used interchangeably. An assurance case is defined as: "*A reasoned and compelling argument, supported by a body of evidence, that a* *system, service or organisation will operate as intended for a defined application in a defined environment"* [40]. It is also defined as: *A collection of auditable claims, arguments, and evidence created to support the contention that a defined system/service will satisfy the particular requirements* [41]. As observed from the former or latter definitions, the term 'assurance case' is generic and does not necessarily indicate safety as the property to be assured. Hence, the term 'assurance case' by its own has no particular focus, but if safety is the intended property to be assured, then using terms such as 'safety case' or 'safety assurance case' is more precise where both can be thought of as an instance of an assurance case.

Although the term 'safety case' has become popular today in many safety critical system domains, but its precise meaning is dependent on the purpose that the safety case is intended to satisfy [3]. This raises the question of: *Why do industries need a safety case?* During this work, different purposes that safety cases can satisfy are observed. A safety case is built as a tool:

- To manage residual risks [42]
- To record engineering practices [3]
- In a court of law to address and reduce legal liability [9, 3]
- For marketing
- Etc.

However, before any safety case is attempted, the rationale and purpose of it must be clearly understood. This is vitally important, because if the specific requirements for compiling a safety case are not clear, then the following safety case will also be not clear [3].

There are different definitions of safety case [43, 3]. Most of the available definitions indicate the consensus that a safety case is oriented to demonstrate how a system reduces risk of specific losses to an acceptable level and thus enable a regulator to assess whether the system is acceptably safe to operate. It is worth pointing out that the definition of safety case by the UK Defence Standard 00-56 [44] is the most common. The standard defines the safety case as:

"*A structured argument, supported by evidence, intended to justify that a system is acceptably safe for a specific application in a specific operating environment*".

$$\text{Safety Argument} \in \text{Safety Case}$$

The work presented in the two parts of this thesis assumes that the main purpose of a safety case is to justify safety and it refers to the safety case definition by the UK Defence Standard 00-56 wherever the term 'safety case' appears.

A safety case comprises elements as follows:

- *Safety requirements or objectives* that are mainly derived to eliminate or mitigate hazards (also known as goals)
- *Lifecycle artefacts* (also know as work products) which are basically the results of each development phase (e.g. safety analyses, software inspections, or functional tests)
- a *Safety argument* explaining how safety goals (in form of safety claims) are supported by available artefacts (in form of safety evidence)
- *Context* and *Assumptions* about the operating environment and usage

Figure 2.3 shows an overview of the safety case elements and the relationships between them.

2.4.3 Safety Argument

The main purpose of a safety case is to communicate an argument. The argument demonstrates how someone can reasonably conclude that a system is acceptably safe from the evidence available [45]. In English, the word 'argument' is defined as: *"A reason or set of reasons that somebody uses to show that something is true or correct"* [20]. A more technical definition for the word 'argument' is: *A body of information presented with the intention to establish one or more claims through the presentation of related supporting claims, evidence, and contextual information.* [41]. An argument in the safety case definition is called a 'safety argument' or 'safety case argument' and it can be defined as a hierarchically connected series of claims supported by evidence. Safety arguments are intended to demonstrate to the reader that a system is acceptably safe as an overall claim. The claim is defined as: *A proposition being asserted by the author or utterer that is a true or false statement* [41]. The evidence is defined as: *Information or objective artifacts being offered in support of one or more claims* [41].

In order for safety cases to be developed, discussed, challenged, presented and reviewed amongst stakeholders, as well as maintained throughout the product lifecycle, it is necessary for the (1) argument to be clearly structured

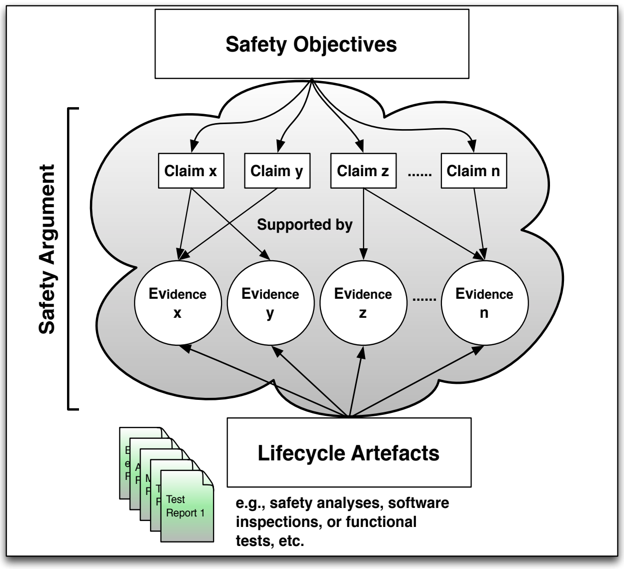


Figure 2.3: Overview of a safety case and its elements

and (2) items of evidence to be clearly asserted to support the argument [40]. There are several ways to represent safety arguments. Safety arguments might be represented by:

- **Prose:** Safety arguments are typically communicated in existing safety cases through free text [45]. Perhaps this is the easiest way to represent safety arguments but not necessarily the most efficient. There are several problems observed while reviewing real safety cases written in prose. We describe some of them. Noticeably, not all engineers who are involved in writing a safety case can write clear and well-structured English [45]. An instance of this problem is, probably, the unconscious use of the ellipsis process in natural languages when authors, unconsciously, leave some non-described crucial details in some statements because they assume that the readers are aware of the context of these statements. For example, the following text describes an evidence item used to support some claims about a bug tracking system of software failures:

"Here we provide evidence of bug tracking for the software. 'XXXXX' is the database that is used to track all issues regarding this system. It has full visibility and is extremely detailed." What does 'all issues' means? Is it the safety, software or bug issues? How about 'visibility'? Does the writer mean the visibility of the software or the visibility of the bug information? [3]

  There are more problems in the description above but the idea is to give an example of the text quality problem.

  Another problem observed in safety cases written in prose is the cross-references among texts. Multiple cross-references in texts can be awkward and disrupt the flow of the main argument [45].

- **Tabular notations:** This way to demonstrate safety arguments is not common. The idea, however, is to arrange an argument claim together with its supportive items of evidence in rows and columns.

- **Graphical notations:** This way represents the individual elements of safety arguments (e.g., safety goals, items of evidence, assumptions, etc.) using graphical symbols (e.g., squares, circles, parallelograms, etc.). The Goal Structuring Notation (GSN), as well as, the Claims Argument Evidence (CAE) notation are two examples of this way.

Discussing the advantages and disadvantages of the three ways listed above is not an objective of this thesis. We do not claim that a problem in one way can not apply to other ways. However, we use the graphical notation since it can clearly represent the elements of safety arguments and their relationships. Moreover, almost all of the related works to this thesis use GSN thus adopting GSN can make the discussions, comparisons and explanations of our work more clear with respect to other works.

The Goal Structuring Notation (GSN)

GSN is a graphical argument notation which can be used to document explicitly the elements and structure of an argument and the argument's relationship to evidence [40]. GSN's notations are used as a means for communicating (1) safety argument elements, claims, argument logic, assumptions, context, evidence and (2) the relationships between these elements [4].

A goal structure shows how goals are successively ***solved by*** sub-goals until a point is reached where claims can be supported by direct reference to evidence. Using GSN, it is also possible to clarify the argument strategies adopted (i.e., how the premises imply the conclusion), the rationale for the approach (assumptions, justifications) and the context in which goals are stated [4].

In GSN, rectangles are used to present the argument's claims (***Goals*** in GSN). Parallelograms is used to present the argument's logic (***Strategies*** in GSN). Circles are used to present items of evidence (***Solutions*** in GSN). Ovals with the letter 'J' at the bottom-right are used to present a statement of rationale (***Justifications*** in GSN). Ovals with the letter 'A' at the bottom-right are used to present an intentionally unsubstantiated statement (***Assumptions*** in GSN) [40]. Squashed rectangles are used to present a reference to contextual information or a statement (***Context*** in GSN). Hollow diamonds are applied to the centre of an element (e.g., goal, assumptions, context, etc.) to indicate that a line of argument has not been developed (***Undeveloped*** ***<element name>*** in GSN) [40]. ***SupportedBy*** is an evidential relationship which declares the link between a goal and the evidence used to substantiate it [40]. Permitted supported by connections are: goal-to-goal, goal-to-strategy, goal-to-solution, strategy-to-goal. ***InContextOf*** is a link that declares a contextual relationship [40]. Permitted connections are: goal-to-context, goal-to-assumption, goal-to-justification, strategy-to-context, strategy-to-assumption and strategy-to-justification [40].

Figure 2.4 shows the principal GSN elements, and Figure 2.5 shows an example of a safety argument represented by those elements.

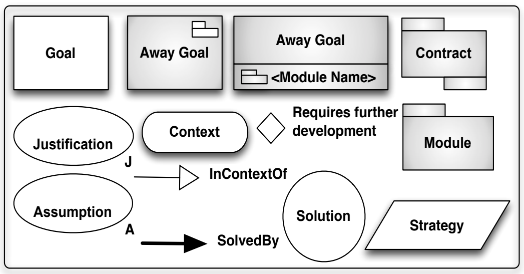


Figure 2.4: Overview of the GSN principal elements

GSN has been extended to enable modularity in a safety case (i.e., module-based development of safety cases). Hence, modular GSN enables the partitioning of a safety case into an interconnected set of modules [18].

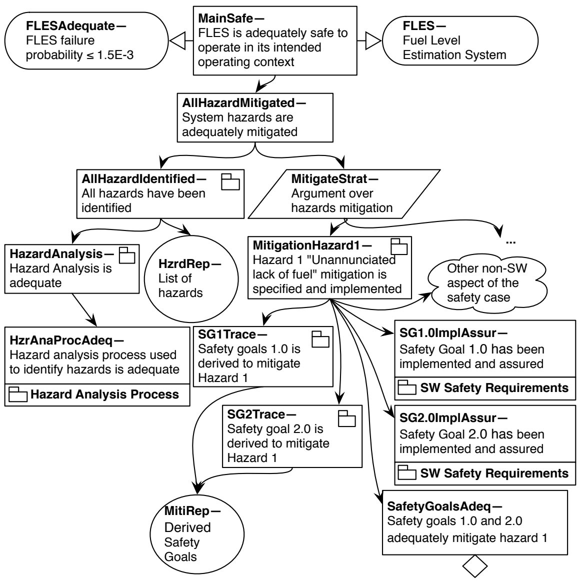

FLESAdequate—
FLES failure probability ≤ 1.5E-3
MainSafe—
FLES is adequately safe to operate in its intended operating context
FLES—
Fuel Level Estimation System
AllHazardMitigated—
System hazards are adequately mitigated
AllHazardIdentified—
All hazards have been identified
MitigateStrat—
Argument over hazards mitigation
...
HazardAnalysis—
Hazard Analysis is adequate
HzrdRep—
List of hazards
MitigationHazard1—
Hazard 1 "Unannunciated lack of fuel" mitigation is specified and implemented
Other non-SW aspect of the safety case
HzrAnaProcAdeq—
Hazard analysis process used to identify hazards is adequate
Hazard Analysis Process
SG1Trace—
Safety goals 1.0 is derived to mitigate Hazard 1
SG1.0ImplAssur—
Safety Goal 1.0 has been implemented and assured
SW Safety Requirements
SG2Trace—
Safety goal 2.0 is derived to mitigate Hazard 1
SG2.0ImplAssur—
Safety Goal 2.0 has been implemented and assured
SW Safety Requirements
MitiRep—
Derived Safety Goals
SafetyGoalsAdeq—
Safety goals 1.0 and 2.0 adequately mitigate hazard 1

Figure 2.5: A safety argument example represented by GSN [46]

Figure 2.4 presents the principal notations of GSN after the extension in gray. An ***Away Goal*** with a bisecting line in the lower half of it the repeats a claim presented in another argument module which is used to support the argument in the local module [40]. The Module Identifier provides a reference to the module that presents the original claim. A ***Module*** reference presents a reference to a module containing an argument. A ***Contract*** module reference presents a reference to a contract module containing definition of the relationships between two modules, defining how a claim in one supports the argument in the other [40].

2.5 Safety Contracts

The term ‘contract’ is defined in English as: *“A written or spoken agreement, especially one concerning employment, sales, or tenancy, that is intended to be enforceable by law”* [20]. A contract is intended to (1) establish a binding relationship between one party’s offer and the acceptance of that offer by one or more parties, and (2) set out the terms and conditions that constrain this relationship. Using the contracts is familiar in software development. For instance, Design by Contract (DbC) was introduced by Meyer [47, 48] to constrain the interactions that occur between objects. Moreover, contract-based design is an approach where the design process is seen as a successive assembly of components where a component behaviour is represented in terms of assumptions about its environment and guarantees about its behavior [49].

In 1969, Hoare introduced the pre- and postcondition technique to describe the connection (dependency) between the execution results (R) of a program (Q) and the values taken by the variables (P) before that program is initiated [50]. Hoare introduced a new notation to describe this connection, as follows:

$$P\ \{Q\}\ R.$$

This notation can be interpreted as: *"If the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion"* [50].

In the context of contract-based design, a contract is conceived as an extension to the specification of software component interfaces that specifies preconditions and postconditions to describe what properties a component can offer once the surrounding environment satisfies one or more related assumption(s).

A contract is said to be a *safety contract* if it guarantees a property that is traceable to a hazard. There have been significant works that discuss how to represent and to use contracts [10, 11, 12]. In the safety critical systems domain, researchers have used, for example, assume-guarantee contracts to propose techniques to lower the cost of developing software for safety critical systems. Moreover, contracts have been exploited as a means for helping to manage system changes in a system domain or in its corresponding safety case [13, 14, 15].

The following is an example that depicts the most common used form of contracts:

Guarantee: The WCET of task X is ≤ 10 milliseconds
Assumptions:
X is:

1. compiled using compiler $[C]$,
2. executed on microcontroller $[M]$ at 1000 MHz with caches disabled, and
3. not interrupted

In this thesis, we distinguish between safety contracts within the system domain and safety argument contracts in the safety case. The former type of contracts captures the dependencies among the system's components. However, a safety argument contract captures the dependencies among the safety case modules. More specifically, a safety argument contract describes the connection between a *consumer* goal in one safety case module and a *provider* goal in another module [40].

Chapter 3

Problem Description and Research Goals

3.1 Problem Description

Safety assurance and certification are amongst the most expensive and time-consuming tasks in the development of safety-critical embedded systems [51]. A key reason behind for this is the increasing complexity and size of these systems combined with their growing market demands. The cost of system changes including the cost of the activities that will follow them, such as regression testing, are another key reason that exacerbate the problems of cost and time in safety certification. Changing regulatory requirements, additional safety evidence and a changing design challenge the corresponding safety case and make safety case maintenance a costly and time-consuming activity. Coherent strategies are required to reduce the cost and time of safety certification.

One of the biggest challenges that affects safety case revision and maintenance is that a safety case documents a complex reality that comprises a complex web of interdependent elements. That is, safety goals, evidence, argument, and assumptions about operating context are highly interdependent. Hence, seemingly minor changes may have a major impact on the contents and structure of the safety argument. Basically, operational or environmental changes may invalidate a safety case argument for two main reasons as follows:

1. Evidence is valid only in the operational and environmental context in which it is obtained, or to which it applies. During or after a system

change, evidence might no longer support the developers' claims because it could reflect old development artefacts or old assumptions about operation or the operating environment.

2. Safety claims, after introducing a change, might be nonsense, no longer reflect operational intent, or be contradicted by new data. Changing safety claims might change the argument structure.

In order to deal with problems that impede safety cases maintenance, we start by identifying and describing these problems.

Main Problem: *Maintaining safety cases after implementing a system change is a painstaking process.* This main problem is caused by three sub-problems.

Sub-problem (1): *The lack of documentation of dependencies among the safety cases contents.*

Developers of safety cases are experiencing difficulties in identifying the direct and indirect impact of change due to high level of dependency among safety case elements. If developers do not understand the impact of change then they have to be conservative and do wider verification (i.e., check more elements than strictly necessary) and this increases the maintenance cost. The Goal Structuring Notation (GSN) [40] was introduced to provide a graphical means of communicating (1) safety argument elements: claims (goals), argument logic (strategies), assumptions, context, evidence (solutions), and (2) the relationships between these elements. The use of a goal based structuring approach helped to produce well-structured arguments that clearly demonstrate the relationships between the argument claims and evidence. However, GSN has not solved the problem of documenting dependencies among the safety cases contents. A well-structured argument helps the developers to mechanically propagate the change through the goal structure. However, it does not evaluate whether the suspect elements of the argument are still valid or not, but rather it can bring these elements to the developers' attention [52].

Safety is a system level property; assuring safety requires safety evidence to be consistent and traceable to system safety goals [5]. Moreover, current standards and analysis techniques assume a top-down development approach to system design. One might suppose that a safety argument structure aligned with the system design structure would make traceability clearer. It might, but safety argument structures are influenced by four factors: (1) modularity of evidence, (2) modularity of the system, (3) process demarcation (e.g., the scope of ISO 26262 items [25]), and (4) organisational structure (e.g., who is working on what) [16]. These factors often make argument structures aligned with the system design structure impractical. However, the need to track changes across the whole safety argument is still significant for maintaining the argument regardless of its structure.

As explained in Section 10.2.2, a contract is conceived as an extension to the specification of software component interfaces that specifies preconditions and postconditions to describe what properties a component can offer once the surrounded environment satisfies one or more related assumption(s). Based on this description, safety contracts can be used as a means to record the dependencies among system components. If we assume a one-to-one mapping between a system component and all the claims that are articulated about it, dependencies among safety argument elements can be conceived through the dependencies between components of the corresponding system that are recorded in contracts. In practice, this notion is far from straightforward because it is infeasible to be achieved and impossible to prove the completeness of the generated contracts, and the expected number of contracts will be too large to easily manage.

Sub-problem (2): *The lack of traceability between a system and its safety case.*

We refer to the ability to relate safety argument fragments to system design components as component traceability (through a safety argument). We refer to evidence across a system's artefacts as evidence traceability.

System developers need both top-down and bottom-up impact analysis approaches to maintain safety cases. A top-down approach is dedicated for analysing the impacted artefacts from the system domain down to the safety argument. In contrast, a bottom-up approach is dedicated for analysing impacted elements from the argument to the corresponding artefacts such as a safety analysis report, test results or requirements specification, etc. The lack of systematic and methodical approaches to analysing impact of change is a key reason behind the maintenance difficulties. However, conducting any style of impact analysis requires a traceability mechanism between the system domain and safety arguments.

There has been significant work on how to use safety contracts as a means to establish the required traceability [16]. The guaranteed properties in the contracts can be mapped to safety argument goals. If the derived safety contracts are associated with the corresponding argument elements, any broken contracts will reveal (i.e., highlight) the associated argument elements and thus enabling easier identification for the impacted parts in the argument due to a system change. However, this is not as simple as it first appears because we still do not know which contracts were affected by the change. In other words, how does a change lead to broken contracts?

Predicting system changes before building a safety argument can be useful because it allows the safety argument to be structured to contain the impact of these changes. Hence, anticipated changes may have predictable and traceable consequences that will eventually reduce maintenance effort. Nevertheless, planning the maintenance of a safety case still faces a key problem.

Sub-problem (3): *System changes and their details cannot be fully predicted and made available up front*

Modularity has been proposed as the key element of the 'way forward' in developing systems [53]. For modular systems, it is claimed that the required maintenance efforts to accommodate predicted changes can be less than the required efforts to accommodate arbitrary changes. This is because having a list of predicted changes during the system design phase allows system engineers to contain the impact of each of those changes in a minimal number of system's modules. Furthermore, predicting system changes before building a safety argument can be useful because it allows the safety argument to be structured to contain the impact of these changes. Hence, predicted changes may also have predictable and traceable consequences that will eventually reduce the maintenance efforts. Nevertheless, planning the maintenance of a safety case still faces two key issues: (1) system changes and their details cannot be fully predicted and made available up front, especially, the software aspects of the safety case as software is highly changeable and harder to manage as they are hard to contain and (2) those changes can be implemented years after the development of a safety case [4].

3.2 Research Goals

In this section, we derive the research questions that should address the identified problems as described in Section 3.1. We first identify the goal of our research and revisit it for each research question.

Main Goal: *Facilitating the accommodation of system changes in safety cases to ultimately enhance safety case maintainability.*

We refer to *"Maintainability"* as the ability to repair or replace the impacted elements of a safety case argument, without having to replace still valid elements, to preserve the validity of the argument. The maintainability degree is said to be high whenever the following three activities are done efficiently:

1. Identifying the impacted elements and those that are not impacted.
2. Minimising the number of impacted elements.
3. Reducing the work needed to make the impacted elements valid again.

However, the work in this thesis does not aim to measure the efficiency of achieving the three activities, but rather it strives to enable them and improve on them. In order to achieve the main goal, we should resolve the problems that affect the accommodation of system changes in safety cases in Subsection 3.1. Hence, we have formulated a set of research questions that should be answered by the thesis contributions.

Question 1: *How can the parts in the safety case impacted by a given system change be identified?*

Question 2: *How can traceability between the system domain and its safety case be established to highlight the impacted parts in one side whenever the other side changes?*

Question 3: *What role can safety contracts play in maintaining safety cases and how to derive them?*

Question 4: *How can a system's potential changes be predicted?*

3.3 Related Work

A consortium of researchers and industrial practitioners called the Industrial Avionics Working Group (IAWG) has proposed using modular safety cases as a means of containing the cost of change. IAWG's Modular Software Safety

Case (MSSC) process facilitates handling system changes as a series of relatively small increments rather than occasional major updates. The process proposes to divide the system into a set of blocks [16, 13]. Each block may correspond to one or more software components but it is associated with exactly one dedicated safety case module. Engineers attempt to scope blocks so that anticipated changes will be contained within argument module boundaries. The process establishes component traceability between system blocks and their safety argument modules using Dependency-Guarantee Relationships (DGRs) and Dependency-Guarantee Contracts (DGCs). Part of the MSSC process is to understand the impact of change so that this can be used as part of producing an appropriate argument. The MSSC process, however, does not give details of how to do this. Moreover, the MSSC process is dependent on a list of predicted change scenarios and it is not meant to handle arbitrary changes. The lack of systematic ways to enable better changes prediction might lead to a big limitation to the process. The work in this thesis addresses this issue.

Kelly [54] suggests identifying preventative measures that can be taken when constructing the safety case to limit or reduce the propagation of changes through a safety case expressed in goal-structure terms. For instance, developers can use broad goals (goals that are expressed in terms of a safety margin) so that the these goals might act as barriers to the propagation of change as they permit a range of possible solutions. A safety case therefore, interspersed with such goals at strategic positions in the goal structure could effectively contain “firewalls” to change. Some of these initial ideas concerning change and maintenance of safety cases have been presented in [52]. However, no work was provided to show how these thoughts can facilitate the maintenance of safety cases.

3.4 Research Method

Research is an investigation to find solutions to scientific and social problems through objective and systematic analysis. Research methods are basically all the methods (e.g., theoretical procedures, experimental studies, numerical schemes, statistical approaches, etc.) that are used by a researcher during a research study [55].

In this section, we demonstrate the process of our research and the followed research method. Figure 3.1 describes the process step by step.

Our research work started with a generic problem definition. The main goal was to support ‘composable’ certification of systems or subsystems based

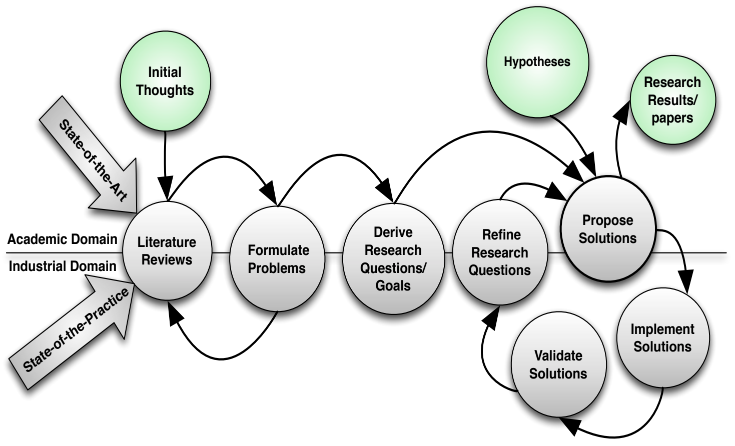


Figure 3.1: Overview of our research process

on reuse of already established arguments and properties of their parts. Even though the idea was generic it formed an initial thought that suggested what and where to look in the literature. Reviewing the literature revealed that the maintenance of safety cases is significant but it seems that it received no or little support yet. Since more clarity of ideas can be acquired through study of literature, we started reviewing the literature with this more specific research goal. To the best of our knowledge neither the state of the art nor the state of the practice set out supporting processes or methods that provide detailed steps of how to analyse the impact of change on safety cases using component or evidence traceability. Subsequently, we started digging deeper and chasing the challenges of safety cases maintenance. After many iterations of the literature survey and problem formulation, we derived the main research goal and other sub-goals. These derived goals motivate us to propose systematic approaches and techniques aimed at reducing the current required efforts for safety cases maintenance due to system changes.

The direct result of our proposed approaches and techniques is a series of research papers that communicates our work to the research community. We use hypothetical and real-world systems to demonstrate our proposed approaches and techniques. Using such systems helps us to find possible limitations and refine our research questions accordingly. However, we have not yet validated our work based on empirical methods thus we did not refine the research questions with respect to validation results. We consider this as a main part of our future work.

Chapter 4

Thesis Contributions

In this section, we present the contributions of the included papers (in ***Part II***). We also show how the contribution of each paper contributes as an answer to one ore more of the formulated research questions in Section 3.2.

4.1 Contributions of the Included Papers

- ***Paper A:*** *An Approach to Maintaining Safety Case Evidence After a System Change*

  The paper proposes a new approach to facilitating safety case change impact analysis. In the approach, automated analysis of information given as annotations to the safety argument highlights suspect safety evidence to bring it to engineers' attention. The approach facilitates identifying the evidence impacted by change by storing additional information in the safety argument. The paper also proposes annotating each reference to a development artefact (e.g. an architecture specification) in a goal or context element with an *artefact version number*. Each solution element will be also annotated with a set of other extra information.

- ***Paper B:*** *Facilitating the Maintenance of Safety Cases*

  The paper shows how to apply the Modular Software Safety Case (MSSC) process to a real safety critical system to show how system engineers can identify the elements in a safety argument that might be impacted by a change. The paper extends safety contracts that were proposed by the MSSC process to include the additional information (From Paper A). The safety contracts were associated with the safety argument using the GSN context notations. This association established a traceability between the system design and its safety case and thus it provided a starting point for the impact analysis in the safety argument. Moreover, the paper shows how extending the safety contract helped to (1) highlight the affected argument elements and (2) identify inadequacies in the generated artefacts from the development lifecycle.

- ***Paper C:*** *Deriving Safety Contracts to Support Architecture Design of Safety Critical Systems*

  The paper includes building the safety argument before and after introducing a change. The paper also shows how the derived safety contracts for parts of the system design can be associated with the corresponding argument fragments to establish a traceability between the system and its safety case.

- ***Paper D:*** *Using Sensitivity Analysis to Facilitate The Maintenance of Safety Cases*

  The paper combines sensitivity analysis together with the concept of contracts to identify the sensitive parts of a system and highlight these parts to help the experts to make an educated decision as to whether or not apply changes. Also, since considering a complete list of anticipated changes is difficult, the paper shows how to determine the flexibility (or compliance) of each component to changes. This means that regardless of the type of changes the latter will be seen as factors to increase or

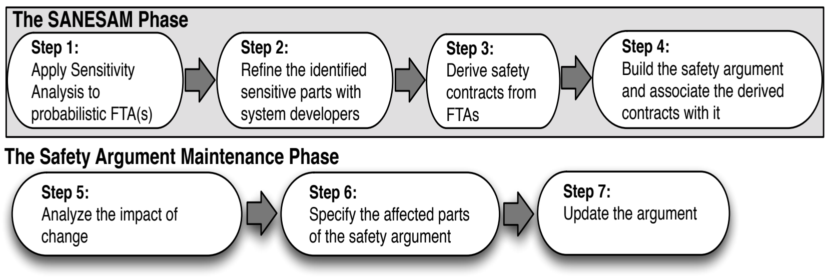

The SANESAM Phase
Step 1: Apply Sensitivity Analysis to probabilistic FTA(s)
Step 2: Refine the identified sensitive parts with system developers
Step 3: Derive safety contracts from FTAs
Step 4: Build the safety argument and associate the derived contracts with it
The Safety Argument Maintenance Phase
Step 5: Analyze the impact of change
Step 6: Specify the affected parts of the safety argument
Step 7: Update the argument

Figure 4.1: Process diagram of the proposed technique

decrease a certain parameter value. Thus system developers can focus more on predicting those changes that might make the parameter value inadmissible. The paper proposes a technique to derive safety contracts from Fault Tree Analysis (FTA) using sensitivity analysis, and a way to map the derived safety contracts to a safety argument to improve the change impact analysis on the safety argument and eventually facilitate its maintenance.

- ***Paper E:*** *Deriving Hierarchical Safety Contracts*

  The main contributions of the paper is to identify some limitations to SANESAM technique which is introduced in Paper D, and suggest two options as extensions to resolve these limitations. The first option is SANESAM+, which is useful in the case of arbitrary changes because it calculates the Failure Probability (FP) for all events in the FTA regardless of any change scenario. The second option is SANESAM+ For Predicted Changes, this option increases the FP for only the events that are associated to a predicted change. A derived safety contract by SANESAM+ For Predicted Changes can guarantee higher FP than the guaranteed FP (for the same event and using the same set of assumptions) in a derived safety contract by SANESAM+. Hence, the derived safety contracts by SANESAM+ For Predicted Changes are more tolerant and robust than those derived by SANESAM+.

4.2 Main Contributions

4.2.1 An Approach to Facilitating Safety Case Change Impact Analysis

This contribution addresses the first research question *"How can the parts in the safety case impacted by a given system change be identified?* In paper A, we provide an approach to facilitate identifying the evidence impacted by a given change. The approach assumes that safety arguments are recorded in GSN and suggests to store additional information in the safety arguments. More specifically, each reference to a development artefact (e.g. an architecture specification) in a goal or context element should be annotated with an *artefact version number*. In addition, each solution element should be annotated with:

1. An *evidence version* number

2. An *input manifest* identifying the inputs (including version) from which the evidence was produced
3. The *lifecycle phase* during which the evidence obtained (e.g. Software Architecture Design)
4. A *safety standard reference* to the clause in the applicable standard (if any) requiring the evidence (and setting out safety integrity level requirements)

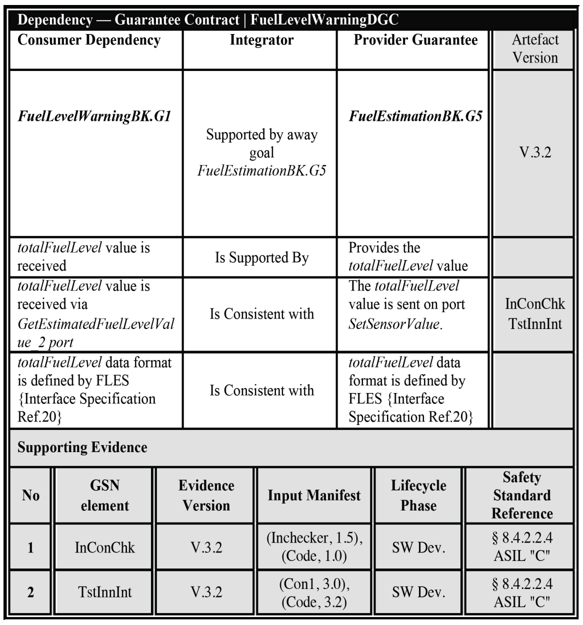
With this data, we can perform a number of automated checks to identify items of evidence impacted by a change. For example:

1. We can determine when two different versions of the same item of evidence are cited in the same argument
2. We can identify out-of-date evidence by searching for input manifests $m = \{(a_1, v_1), ..., (a_n, v_n)\}$ and artefact versions (a, v) such that $\exists i \bullet a = a_i \wedge v > v_i$
3. Where we know a particular artefact has changed, we can search for input manifests containing old versions

In paper B, we extend a state of the art work by this approach. More clearly, we propose storing additional information in the Dependency-Guarantee Contracts (DGCs) of the MSSC process (described in Section 3.3). This additional information can highlight suspect safety evidence and brings it to engineers' attention once it changes. Figure 4.2 shows an extended DGC where the extension is represented by the gray cells. Furthermore, in paper D we recommend using the approach in the derived safety contracts.

4.2.2 A New Safety Contract Notation

This contribution addresses the second research question *"How can traceability between the system domain and its safety case be established to highlight the impacted parts in one side whenever the other side changes?"* We refer to the ability to relate safety argument fragments to system design components as component traceability (through a safety argument). We refer to evidence across a system's artefacts as evidence traceability. In papers A and B, we propose new annotations to the safety argument to support the evidence traceability. Through the annotations, readers of the argument can know the version of an evidence's item, its input manifest, the lifecycle phase in which it was



Dependency — Guarantee Contract | FuelLevelWarningDGC

Consumer Dependency	Integrator	Provider Guarantee	Artefact Version
FuelLevelWarningBK.G1	Supported by away goal *FuelEstimationBK.G5*	***FuelEstimationBK.G5***	V.3.2
totalFuelLevel value is received	Is Supported By	Provides the *totalFuelLevel* value	
totalFuelLevel value is received via *GetEstimatedFuelLevelValue_2 port*	Is Consistent with	The *totalFuelLevel* value is sent on port *SetSensorValue*.	InConChk TstInnInt
totalFuelLevel data format is defined by FLES {Interface Specification Ref.20}	Is Consistent with	*totalFuelLevel* data format is defined by FLES {Interface Specification Ref.20}	

Supporting Evidence

No	GSN element	Evidence Version	Input Manifest	Lifecycle Phase	Safety Standard Reference
1	InConChk	V.3.2	(Inchecker, 1.5), (Code, 1.0)	SW Dev.	§ 8.4.2.2.4 ASIL "C"
2	TstInnInt	V.3.2	(Con1, 3.0), (Code, 3.2)	SW Dev.	§ 8.4.2.2.4 ASIL "C"

Figure 4.2: An extended DGC

obtained, and the reference of the safety standard recommendations (if applicable). Using the annotations can be useful to identify the suspect goals, their arguments, as well as, the supporting evidence.

In papers B and C, we derive safety contracts that contain guarantee and assumptions about system design components. The guarantees and assumptions are represented as GSN goals while arguing about those design components. In order to enable efficient traceability between safety argument fragments and system design components, we use the GSN context notation. We simply create contexts that refer to a specific safety contract name and associate them to appropriate GSN goals to indicate that these goals are part of the associated contract. GSN's standard [40] states that context notations are used to present a reference to contextual information or a statement. However, the standard offers no normative model of how context affects the meaning of arguments. To not misuse GSN context elements and to avoid any confusion or misinterpretation, we propose a new notation to indicate safety contracts both in the safety argument and in the system artefacts. Figure 4.3 show our proposed

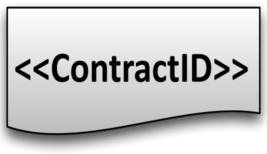


Figure 4.3: A new safety contract notation

safety contract notation. It is worth noting that neither the annotations nor the safety contract notation shall affect the way GSN is being produced but it brings additional information for developers' attention.

4.2.3 Sensitivity Analysis for Enabling Safety Argument Maintenance (SANESAM)

In papers D and E, we propose a technique the utilises sensitivity analysis to identify the sensitive parts of a system and records informations about these parts using safety contracts. The main objective of the technique is to help the experts to make an educated decision as to whether or not apply changes. This decision is in light of beforehand knowledge of the impact of these changes on the system and its safety case. More clearly, using the technique helps to (1) bring to developers' attention the most sensitive parts of a system for a particular change and (2) manage the change by guiding the developers to the parts in the safety argument that might be affected after applying a change. We do not claim that using safety contracts as a way of managing change is a new notion since it has been discussed in some works, such as [16][13], but deriving the contracts and their contents is one of the technique's objectives. Proposing the technique addresses our third research question: *"What role can safety contracts play in maintaining safety cases and how to derive them?"*

The technique comprises 7 steps that are distributed between the Sensitivity ANalysis for Enabling Safety Argument Maintenance (SANESAM) phase and the safety argument maintenance phases as shown in Figure 4.1. The steps of SANESAM phase are represented along the upper path, whilst the lower path represents the steps of the safety argument maintenance phase. The SANESAM phase, however, is what is being discussed in this specific contribution. The rationale of SANESAM is to determine, for each component, the allowed range for a certain parameter within which a component may change before it compromises a certain system property (e.g., safety, reliability, etc.).

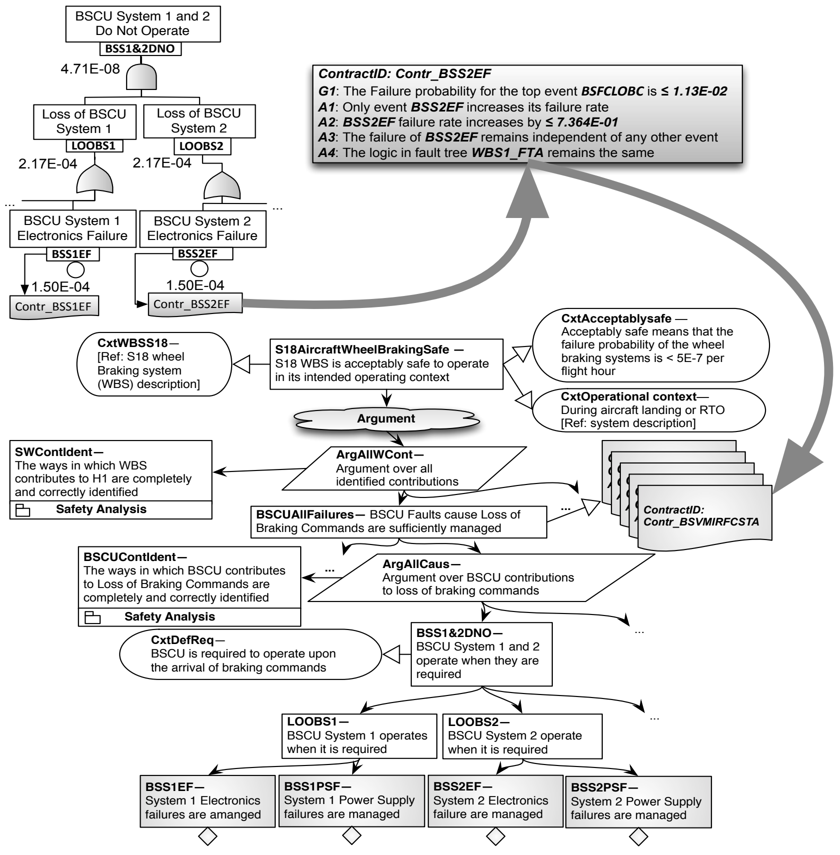

BSCU System 1 and 2 Do Not Operate
BSS1&2DNO
4.71E-08
Loss of BSCU System 1
LOOBS1
Loss of BSCU System 2
LOOBS2
2.17E-04
2.17E-04
BSCU System 1 Electronics Failure
BSS1EF
BSCU System 2 Electronics Failure
BSS2EF
1.50E-04
1.50E-04
Contr_BSS1EF
Contr_BSS2EF
ContractID: Contr_BSS2EF
G1: The Failure probability for the top event BSFCLOBC is ≤ 1.13E-02
A1: Only event BSS2EF increases its failure rate
A2: BSS2EF failure rate increases by ≤ 7.364E-01
A3: The failure of BSS2EF remains independent of any other event
A4: The logic in fault tree WBS1_FTA remains the same
CxtWBSS18— [Ref: S18 wheel Braking system (WBS) description]
S18AircraftWheelBrakingSafe — S18 WBS is acceptably safe to operate in its intended operating context
CxtAcceptablysafe — Acceptably safe means that the failure probability of the wheel braking systems is < 5E-7 per flight hour
CxtOperational context— During aircraft landing or RTO [Ref: system description]
Argument
SWContIdent— The ways in which WBS contributes to H1 are completely and correctly identified
Safety Analysis
ArgAllWCont— Argument over all identified contributions
ContractID: Contr_BSVMIRFCSTA
BSCUAllFailures— BSCU Faults cause Loss of Braking Commands are sufficiently managed
BSCUContIdent— The ways in which BSCU contributes to Loss of Braking Commands are completely and correctly identified
Safety Analysis
ArgAllCaus— Argument over BSCU contributions to loss of braking commands
CxtDefReq— BSCU is required to operate upon the arrival of braking commands
BSS1&2DNO— BSCU System 1 and 2 operate when they are required
LOOBS1— BSCU System 1 operates when it is required
LOOBS2— BSCU System 2 operate when it is required
BSS1EF— System 1 Electronics failures are amanged
BSS1PSF— System 1 Power Supply failures are managed
BSS2EF— System 2 Electronics failure are managed
BSS2PSF— System 2 Power Supply failures are managed

Figure 4.4: Illustration of the role of safety contracts in FTA and safety arguments

To this end, we use sensitivity analysis as a method to determine the range of failure probability parameter for each component. Hence, the technique assumes the existence of a probabilistic FTA where each event in the tree is specified by an actual (i.e., current) failure probability $FP_{Actual|event(x)}$. In addition, the technique assumes the existence of the required failure probability for the top event $FP_{Required(Topevent)}$, where the FTA is considered unacceptable if: $FP_{Actual(Topevent)} > FP_{Required(Topevent)}$.

The technique derives safety contracts for the identified sensitive parts. The main objective of the contracts is to (1) highlight the sensitive events to make them visible up front for developers attention and (2) to record the dependencies between the sensitive events and the other events in the FTA. Hence, if any contracted event has received a change that necessitates increasing its failure probability where the increment is still within the defined threshold in the contract, then it can be said that the contract(s) in question still holds (intact) and the change is containable with no further maintenance. The contract(s), however, should be updated to the latest failure probability value. On the contrary, if the change causes a bigger increment in the failure probability value than the contract can hold, then the contract is said to be broken and the guaranteed event will no longer meet its reliability target.

Figure 4.4 shows how the technique derives safety contracts from FTA and how these contracts are associated to the safety arguments' fragments.

4.2.4 Support the Prediction of Potential System Changes

Expectedly, if we ask system engineers to anticipate the potential future changes for a system they might brainstorm and come up with a list of changes. However, the list can be incomplete or contain unlikely changes that might influence the system design to little or no avail. Instead, we propose providing system developers a list of system parts that may be more problematic to change than other parts and ask them to choose the parts that are most likely to change. Of course our list can be augmented by additional changeable parts that may be provided by the system developers. This contribution addresses our fourth research question: *"How can a system's potential changes be predicted?"* This contribution can be represented by the first two steps of SANESAM as shown in Figure 4.1, as follows:

- ***Step 1. Apply the sensitivity analysis to a probabilistic FTA***: In this step the sensitivity analysis is applied to a FTA to identify the sensitive events whose minimum changes have the maximal effect on the $FP_{Topevent}$. Identifying those sensitive events requires the following steps to be performed:

  1. Find minimal cut set MC in the FTA.
  2. Calculate the maximum possible increment in the failure probability parameter of event x before the top event $FP_{Actual(Topevent)}$ is no longer met, where $x \in MC$ and

$$(FP_{Increased|event(x)} - FP_{Actual|event(x)}) \not\Rightarrow$$
$$FP_{Actual(Topevent)} > FP_{Required(Topevent)}.$$

3. Rank the sensitive events from the most sensitive to the less sensitive. The most sensitive event is the event for which the following equation is the minimum:

$$(FP_{Increased|event(x)} - FP_{Actual|event(x)})/FP_{Actual|event(x)}$$

- ***Step 2. Refine the identified sensitive parts with system developers***: In this step, the generated list from Step 1 should be discussed with system developers (e.g., safety engineers) and ask them to choose the sensitive events that are most likely to change. The list can be extended to add any additional events by the developers. Moreover, it is envisaged that some events may be removed from the list or the rank of some of them change.

Chapter 5

Conclusions and Future Work

Evidence might be invalidated by changes to the system design, operation, or environmental context. Assumptions valid in one context might be invalid elsewhere. The impact of change might not be obvious. This thesis proposes a new method to facilitate safety case maintenance by highlighting the impact of changes. Moreover, changes are often only performed years after the initial design of the system making it hard for the designers performing the changes to know which parts of the argument are affected. Using contracts to manage system changes is not a novel idea; there has been significant work discusses how to represent contracts and how to use them. However, there has been little work on how to derive them. In this thesis, we propose a technique that uses sensitivity analysis to support the prediction of future changes and to derive safety contracts. We also propose a way to map the derived safety contracts to a safety argument to improve the change impact analysis on the safety argument and eventually facilitate its maintenance. In this chapter, we provide the concrete contributions of this thesis as a conclusion and we also suggest possible future research directions.

5.1 Conclusions

The main goal of this thesis is to facilitate the accommodation of system changes in safety cases to ultimately enhance safety case maintainability. A complete approach to managing safety case change would include (a) a set of predicted change scenarios, (b) mechanisms to structure the argument so as to contain the impact of predicted changes, and (c) means of assessing the impact of change on all parts of the argument. The main contributions of the thesis are:

1. The thesis proposes a technique that uses sensitivity analysis to facilitate the maintenance of a safety case. The technique comprises 7 steps that are distributed between the Sensitivity ANalysis for Enabling Safety Argument Maintenance (SANESAM) phase and the safety argument maintenance phases as shown in Figure 4.1. The steps of the SANESAM phase are represented along the upper path, whilst the lower path represents the steps of the safety argument maintenance phase. SANESAM (steps 1-4 in Figure 4.1) is the main focus of the thesis. We use sensitivity analysis to support change prediction and prioritisation. We also use safety contracts to record the information about changes that will ultimately advise the engineers what to consider and check when changes actually happen **(Paper D and E)**.

2. The thesis proposes an extension to the safety contracts that were introduced by the MSSC process to include additional information that facilitates impact analysis (**Paper B**). Additionally, The thesis proposes a method to associate derived safety contracts of system components with the corresponding parts in the safety arguments **(Paper C)**.

3. The thesis proposes a new approach to facilitating safety case change impact analysis. In (**paper A and B**), automated analysis of information given as annotations to the safety argument highlights suspect safety evidence to bring it to engineers' attention. This contribution deals with steps 5 and 6 in Figure 4.1.

The relationship between the included papers and the identified research questions in Section 3.2 is summarised in Table 5.1.

	Question 1	Question 2	Question 3	Question 4
Paper A	✓			
Paper B	✓	✓		
Paper C		✓		
Paper D		✓	✓	✓
Paper E		✓	✓	✓

Table 5.1: Thesis contributions to research questions

5.2 Future Work

In this section we present some possible future work as follows:

- **Extend and automate our approach of facilitating safety case change impact analysis**: In Section 4.2.1, we discussed an approach to facilitating safety case change impact analysis as a contribution to this thesis. We have not considered the full range of properties that we could check with automated analyses or the annotations necessary to support those analyses. We have likewise not yet studied the feasibility or value of such automated checks by implementing and applying them. We leave these efforts to future work.

- **Describe the second part of the sensitivity based technique:** In Section 4.2.3, we discussed the first part of the technique as a contribution to this thesis. As a future plan, we want to describe the second part of the technique (the last three steps of the technique as shown in Figure 4.1). In addition, we plan to create a case study to validate both the feasibility and efficacy of SANESAM and the second part of the technique.

Bibliography

[1] J.C. Knight. Safety critical systems: Challenges and directions. In *Proceedings of the 24rd International Conference on Software Engineering (ICSE).*, pages 547–550, May 2002.

[2] Anthony Hidden (QC). *Investigation into the Clapham Junction Railway Accident*. HMSO, 1989.

[3] R. Maguire. *Safety Cases and Safety Reports: Meaning, Motivation and Management*. Ashgate Publishing, Ltd., 2012.

[4] O. Jaradat, I. Bate, and S. Punnekkat. Using sensitivity analysis to facilitate the maintenance of safety cases. In *Proceedings of the 20th International Conference on Reliable Software Technologies (Ada-Europe)*, pages 162–176, June 2015.

[5] T. Kelly and J. McDermid. A systematic approach to safety case maintenance. In *Proceedings of the Computer Safety, Reliability and Security*, volume 1698 of *Lecture Notes in Computer Science*, pages 13–26. Springer Berlin Heidelberg, 1999.

[6] U.K. Ministry of Defence, “JSP 430 - Ship Safety Management System Handbook”, Ministry of Defence January 1996.

[7] T. Kelly. *Arguing Safety – A Systematic Approach to Managing Safety Cases*. PhD thesis, Department of Computer Science, University of York, 1998.

[8] Health and Safety Executive (HSE). *Railway Safety Cases - Railway (Safety Case) Regulations - Guidance on Regulations*, 1994.

[9] Jacobs Sverdrup Australia Pty, Ltd. The development of safety cases for complex safety critical systems. lecture notes. April 2005. [online]. avaialble: `https://msquair.files.wordpress.com/2012/06/md12_safety_cases_r5.pdf`.

[10] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis. Multiple viewpoint contract-based specification and design. In *Proceedings of the 6th International Symposium, FMCO*, pages 200–225, Amsterdam, The Netherlands, October 2007. Springer.

[11] W. Damm, H. Hungar, J. Bernhard, T. Peikenkamp, and I. Stierand. Using contract-based component specifications for virtual integration testing and architecture design. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2011.

[12] S. Bauer, A. David, R. Hennicker, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski. Moving from specifications to contracts in component-based design. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering*, FASE'12, pages 43–58, Berlin, Heidelberg, 2012. Springer-Verlag.

[13] J. L. Fenn, R. Hawkins, P. J. Williams, T. Kelly, M. G. Banner, Y. Oakshott. The who, where, how, why and when of modular and incremental certification. In *Proceedings of the 2nd IET International Conference on System Safety*, pages 135–140. IET, 2007.

[14] P. Conmy, J. Carlson, R. Land, S. Björnander, O. Bridal, I. Bate. Extension of techniques for modular safety arguments. Deliverable d2.3.1, technical report, Safety certification of software-intensive systems with reusable components (SafeCer), 2012.

[15] P. Graydon and I. Bate. The nature and content of safety contracts: Challenges and suggestions for a way forward. In *Proceedings of the 20th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, November 2014.

[16] Modular Software Safety Case (MSSC) — Process Description. [online]. available: https://www.amsderisc.com/related-programmes, Nov 2012.

[17] O. Jaradat, P. Graydon and I. Bate. An approach to maintaining safety case evidence after a system change. In *Proceedings of the 10th European Dependable Computing Conference (EDCC)*, Newcastle, UK, August 2014.

[18] O. Jaradat, I. Bate and S. Punnekkat. Facilitating the maintenance of safety cases. In *Proceedings of the 3rd International Conference on Reliability, Safety and Hazard - Advances in Reliability, Maintenance and Safety (ICRESH-ARMS)*, Luleå, Sweden, June 2015.

[19] I. Ŝljivo, O. Jaradat, I. Bate, and P. Graydon. Deriving safety contracts to support architecture design of safety critical systems. In *Proceedings of the 16th IEEE International Symposium on High Assurance Systems Engineering*, pages 126–133. IEEE, January 2015.

[20] *Oxford Dictionary of English (3 ed.)*. Oxford University Press, 2010.

[21] Modular Software Safety Case Process, Glossary Doc 202: (https://www.amsderisc.com/related-programmes/); Copyright 2012 (C) AgustaWestland Limited, BAE SYSTEMS, GE Aviation, General Dynamics United Kingdom Limited, and SELEX Galileo Ltd. Nov 2012.

[22] J. Knight. *Fundamentals of Dependable Computing for Software Engineers*. Chapman & Hall/CRC, 1st edition, 2012.

[23] M. Dorfman and C. Anderson. *Aerospace software engineering: a collection of concepts*. American Institute of Aeronautics and Astronautics, Washington, DC, 1991.

[24] I. Sommerville. *Software Engineering*. Addison-Wesley, Harlow, England, 9 edition, 2010.

[25] ISO 26262:2011. Road Vehicles — Functional Safety, Part 1-9. International Organization for Standardization, Nov 2011.

[26] A. Aviẑienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan 2004.

[27] N. Leveson. Software system safety, Lecture Notes, Massachusetts Institute of Technology (MIT), Aero/Astro Department, 2002.

[28] M. Rausand. *Reliability of Safety-Critical Systems: Theory and Applications*. John Wiley and Sons, Inc., Hoboken, New Jersey, 2013.

[29] M. Rausand and A. Høyland. *System Reliability Theory: Models, Statistical Methods and Applications*. Wiley-Interscience, Hoboken, NJ, 2004.

[30] SAE ARP4761 Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, December 1996.

[31] M. Stamatelatos, W. Vesely, J. Dugan, J. Fragola, J Minarick, and J. Railsback. Fault Tree Handbook with Aerospace Applications. Handbook, National Aeronautics and Space Administration, Washington, DC, 2002.

[32] H. E. Lambert. Use of fault tree analysis for automotive reliability and safety analysis. SAE technical paper, Lawrence Livermore National Lab., 2004.

[33] A. Saltelli. *Global sensitivity analysis: the primer.* John Wiley, 2008.

[34] L. Breierova and M. Choudhari. An introduction to sensitivity analysis. Technical report, Massachusetts Institute of Technology (MIT), September 1996.

[35] A.C. Cullen and H.C. Frey. *Probabilistic techniques in Exposure assessment.* Plenum Press, New York, 1999.

[36] D. J. Pannell. Sensitivity analysis of normative economic models: theoretical framework and practical strategies. *Agricultural Economics*, 16(2):139 – 152, 1997.

[37] *Nuclear Installations Act 1965, section 14.* Her Majesty's Stationary Office, London, UK, 1965 (reprinted 1993).

[38] K. Cassidy. CIMAH Safety Cases — the HSE Approach. IChemE Symposium series no. 110, 1988.

[39] R B. Whittingham. *Preventing corporate accidents : An Ethical Approach.* Elsevier Ltd, Linacre House, Jordan Hill, Oxford OX2 8DP, UK, 2008.

[40] GSN Community Standard, version 1; (c) 2011 origin consulting (york) limited. http://www.goalstructuringnotation.info.

[41] Object Management Group (OMG). *Structured Assurance Case Metamodel (SACM)*, Technical report, Version 1.0, 2013. [Online]. Available: `http://www.omg.org/spec/SACM/1.0/PDF/`.

[42] P. Graydon. (personal communication), August 2015.

[43] T. Kelly. Introduction to safety cases, lecture notes, 2007. [online]. available: `http://www.omg.org/news/meetings/workshops/SWA_2007_Presentations/00-T3_Kelly.pdf`.

[44] U.K. Ministry of Defence. *00-56 Defence Standard — Safety Management Requirements for Defence Systems*, December 1996.

[45] T. Kelly. A systematic approach to safety case management. In *Proceedings of SAE 2004 World Congress, Detroit*. The Society for Automotive Engineers, March 2004.

[46] O. Jaradat, P. Graydon and I. Bate. The role of architectural model checking in conducting preliminary safety assessment. In *Proceedings of the 31st International System Safety Conference (ISSC)*, Boston, USA, August 2013.

[47] B. Meyer. Design by contract. Technical Report TR-EI-12/CO, Interactive Software Engineering Inc., 1986.

[48] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.

[49] L. Benvenuti, A. Ferrari, E. Mazzi, and A. L. Vincentelli. Contract-based design for computation and verification of a closed-loop hybrid system. In *Proceedings of the 11th International Workshop on Hybrid Systems: Computation and Control*, HSCC '08, pages 58–71, Berlin, Heidelberg, 2008. Springer-Verlag.

[50] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[51] H. Espinoza, A. Ruiz, M. Sabetzadeh, and P. Panaroni. Challenges for an open and evolutionary approach to safety assurance and certification of safety-critical systems. In *Proceedings of the 1st International Workshop on Software Certification (WoSoCER)*, pages 1–6, Nov 2011.

[52] S. Wilson, T. Kelly, and J. McDermid. Safety case development: Current practice, future prospects. In *Proceedings of the 12th Annual CSR Workshop - Software Bases Systems*. Springer-Verlag, 1997.

[53] S. Bates, I. Bate, R. Hawkins, T. Kelly, J. McDermid, and R. Fletcher. Safety case architectures to complement a contract-based approach to

designing safe systems. In *Proceedings of the 21st International System Safety Conference (ISSC)*, 2003.

[54] T. Kelly. Literature survey for work on evolvable safety cases. Department of Computer Sceince, University of York, 1st Year Qualifying Dissertation, 1995.

[55] S. Rajasekar, P. Philominathan, V. Chinnathambi. Research methodology. October 2013. [Online]. Available: http://arxiv.org/pdf/physics/0601009v3.pdf [Lastchecked]: October 2015.

II

Included Papers

Chapter 6

Paper A: An Approach to Maintaining Safety Case Evidence After A System Change

Omar Jaradat, Patrick Graydon, Iain Bate

In Proceedings of the 10th European Dependable Computing Conference (EDCC 2014), Newcastle, UK, May 2014.

Abstract

Developers of some safety critical systems construct a safety case. Developers changing a system during development or after release must analyse the change's impact on the safety case. Evidence might be invalidated by changes to the system design, operation, or environmental context. Assumptions valid in one context might be invalid elsewhere. The impact of change might not be obvious. This paper proposes a method to facilitate safety case maintenance by highlighting the impact of changes.

6.1 Introduction

Developers of some safety critical systems construct a *safety case* comprising both safety evidence (e.g. safety analyses, software inspections, or functional tests) and a *safety argument* explaining that evidence. The safety argument shows which claims the developer uses each item of evidence to support and how those claims, in turn, support broader claims about system behaviour, hazards addressed, and, ultimately, acceptable safety. Changes to the system during or after development might invalidate safety evidence or argument. Evidence might no longer support the developers' claims because it reflects old development artefacts or old assumptions about operation or the operating environment. In the updated system, existing safety claims might be nonsense, no longer reflect operational intent, or be contradicted by new data. To maintain the safety case after the system is changed, developers must analyse the change's impact. This analysis is traditionally done by hand: developers determine whether the evidence still supports the claims made of it, check to see whether new or updated safety requirements are reflected in the argument, and manually review the argument's logic. In this paper, we propose a method to facilitate safety case change impact analysis by automatically highlighting some kinds of impacts.

For the sake of simplicity, we assume in this paper that safety arguments are recorded in the Goal Structuring Notation (GSN) [1]. However, the method we propose might (with suitable adaptations) be suitable for use with other graphical assurance argument notations.

6.2 Our Proposal

A complete approach to managing safety case change would include both (a) mechanisms to structure the argument so as to contain the impact of predicted changes and (b) means of assessing the impact of change on all parts of the argument. In this paper, we focus on identifying the evidence that must be updated to reflect any given change.

To facilitate identifying the evidence impacted by change, we propose storing additional information in the safety argument. We propose annotating each reference to a development artefact (e.g. an architecture specification) in a goal or context element with an *artefact version number*. We also propose annotating each solution element with:

1. An *evidence version* number

2. An *input manifest* identifying the inputs (including version) from which the evidence was produced
3. The *lifecycle phase* during which the evidence obtained (e.g. Software Architecture Design)
4. A *safety standard reference* to the clause in the applicable standard (if any) requiring the evidence (and setting out safety integrity level requirements)

With this data, we can perform a number of automated checks to identify items of evidence impacted by a change. For example:

1. We can determine when two different versions of the same item of evidence are cited in the same argument
2. We can identify out-of-date evidence by searching for input manifests $m = \{(a_1, v_1), ..., (a_n, v_n)\}$ and artefact versions (a, v) such that $\exists i \bullet a = a_i \wedge v > v_i$
3. Where we know a particular artefact has changed, we can search for input manifests containing old versions

If we had further information which inputs were used to produce each input listed in each input manifest, each input that was used to produce those, and so on, we could extend checks (2) and (3) above to *indirect* inputs. For example, suppose that life testing is used to establish the reliability of a component, that this component and its reliability appear in a Failure Modes and Effects Analysis (FMEA), and that the FMEA results are used in a Fault Tree Analysis (FTA). With the additional information, we could compute a closure of the FTA's input manifest that would include the life testing results.

Other analyses may be possible. For example, we suggest storing the safety standard reference to facilitate analysis of impacts that change the safety integrity level of a requirement. However, we have not yet thought these through.

6.3 An Illustrative Example

To illustrate our proposal, consider how the analysis might work on a sample system. Figure 6.1 presents part of an assurance argument we built for a specimen safety critical system we built in prior work [2]. The Fuel Level Estimation System (FLES) is meant to monitor fuel levels to prevent loss of engine power due to running out of fuel. (Running out of fuel is a serious problem in heavy road vehicles because steering and braking mechanisms are powered by the engine; loss of engine power while driving could result in an accident.)

The argument fragment concerns model checking analyses of the system architecture [2]. The FLES architecture, specified in the Architecture Analysis and Design Language (AADL) [3], comprises five threads: `SoftwareIN`, `FuelEstimation`, `FuelLevelWarning`, `Other_Functions` and `SoftwareOUT`. These threads run on a single-core microprocessor with non-preemptive scheduling. Using the UPPAAL model checker, we verified that the system as specified in the architecture is schedulable and free from livelock and deadlock. Rectangular goal element G:LivelocksFree represents the claim that the architecture is free from livelock. G:LivelocksFree's connection to round solution element S:CtrlFloAn shows that this claim is supported by the control flow analysis done using the model checker.

The green elements in Figure 6.1 represent the annotations described in Section 6.2. (These need not necessarily be presented to the user in visual depictions.) Let us consider an example change scenario to illustrate how this information aids safety case change impact analysis. Suppose that the architecture was simplified by removing the `FuelEstimation` thread and moving the tasks it contains to the `FuelLevelWarning` thread. Suppose that an engineer making this change had updated the artefact version annotation(s) in part of the argument referring to the functional behaviour of in those threads. An automated implementation of check (2) described in Section 6.2 could highlight the need to re-run the control flow analysis as well. If the new version of the architecture is version 1.1, analysis of the manifest associated with S:CtrlFloAn would reveal evidence based on an older version of the architecture and tools could flag S:CtrlFloAn as out of date and suspect.

Automated analysis might also highlight goal G:EstimatorArchFree because its artefact version annotation refers to an out-of-date version of the AADL architecture. The goal and its supporting argument are suspect because they might refer to parts of the architecture that no longer exist or make claims about the architecture that are no longer true.

6.4 Related Work

Weaver, McDermid, and Kelly proposed characterising safety evidence according to, amongst other things, the type of technique that produced it (e.g., analysis, testing, inspection, etc.) [4]. Their characterisation was meant to fa-

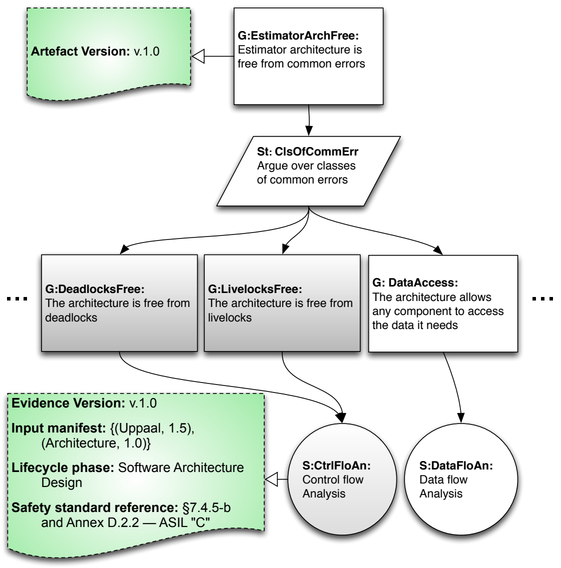

Artefact Version: v.1.0
G:EstimatorArchFree: Estimator architecture is free from common errors
St: ClsOfCommErr Argue over classes of common errors
...
G:DeadlocksFree: The architecture is free from deadlocks
G:LivelocksFree: The architecture is free from livelocks
G: DataAccess: The architecture allows any component to access the data it needs
...
Evidence Version: v.1.0
Input manifest: {(Uppaal, 1.5), (Architecture, 1.0)}
Lifecycle phase: Software Architecture Design
Safety standard reference: §7.4.5-b and Annex D.2.2 — ASIL "C"
S:CtrlFloAn: Control flow Analysis
S:DataFloAn: Data flow Analysis

Figure 6.1: Model Checking Module — Argument Fragment [2].

cilitate judgment of the sufficiency of the evidence. We propose a different characterisation of safety evidence with a different purpose.

Tracking version information and using it to determine when artefacts are out of date is by no means new; `make` does this. Our contribution lies in applying this idea to safety arguments and safety case change impact analysis.

6.5 Conclusion

Maintaining safety arguments after implementing a system change is painstaking process. In this paper we propose a new approach to facilitating safety case change impact analysis. In the approach, automated analysis of information given as annotations to the safety argument highlights suspect safety evidence to bring it to engineers' attention. We illustrated the approach with an example drawn from an automotive system.

We have not considered the full range of properties that we could check with automated analyses or the annotations necessary to support those analyses. We have likewise not yet studied the feasibility or value of such automated checks by implementing and applying them. We leave these efforts to future work.

Acknowledgment

We acknowledge the Swedish Foundation for Strategic Research (SSF) SYNOPSIS Project for supporting this work.

Bibliography

[1] GSN community standard version 1. Technical report, Origin Consulting (York) Limited, November 2011.

[2] Omar Jaradat, Patrick Graydon, and Iain Bate. The role of architectural model checking in conducting preliminary safety assessment. In *Proceedings of the 31st International System Safety Conference (ISSC), System Safety Society*, August 2013.

[3] Peter H. Feiler, David P. Gluch, and John J. Hudak. The architecture analysis & design language (AADL): An introduction. Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, 2006.

[4] R. A. Weaver, J. A. McDermid, and T. P. Kelly. Software safety arguments: Towards a systematic categorisation of evidence. In *Proc. 20 th International System Safety Conference, Denver USA, System Safety Society*, 2002.

Chapter 7

Paper B: Facilitating the Maintenance of Safety Cases

Omar Jaradat, Iain Bate, Sasikumar Punnekkat
In Proceedings of the 3rd International Conference on Reliability, Safety and Hazard — Advances in Reliability, Maintenance and Safety (ICRESH-ARMS 2015), Luleå, Sweden, June 2015.

Abstract

Developers of some safety critical systems construct a safety case comprising both safety evidence, and a safety argument explaining that evidence. Safety cases are costly to produce, maintain and manage. Modularity has been introduced as a key to enable the reusability within safety cases and thus reduces their costs. The Industrial Avionics Working Group (IAWG) has proposed Modular Safety Cases as a means of containing the cost of change by dividing the safety case into a set of argument modules. IAWG's Modular Software Safety Case (MSSC) process facilitates handling system changes as a series of relatively small increments rather than occasional major updates. However, the process does not provide detailed guidelines or a clear example of how to handle the impact of these changes in the safety case. In this paper, we apply the main steps of MSSC process to a real safety critical system from industry. We show how the process can be aligned to ISO 26262 obligations for decomposing safety requirements. As part of this, we propose extensions to MSSC process for identifying the potential consequences of a system change (i.e., impact analysis), thus facilitating the maintenance of a safety case.

7.1 Introduction

Constructing safety cases receives significant industrial attention as it is required for the certification process of many safety critical system domains. A safety case comprises both safety evidence (e.g. safety analyses, software inspections, or functional tests) and a safety argument explaining that evidence. Safety arguments show how system developers use each item of evidence to support claims, and how those claims, in turn, support broader claims about system behaviour, hazards addressed, and, ultimately, acceptable safety [1]. The production, management and evaluation of safety cases are considered difficult to achieve and time consuming. As an anecdotal example, the size of the preliminary safety case for surveillance on airport surfaces with ADS-B [2] is about 200 pages, and it is expected to grow as the operational safety case is created [3]. It is worth noting that a safety case is a living document that grows as the system grows. A safety case should be maintained as needed whenever some aspect of the system, its operation, its operating context, or its operational history changes.

Operational or environmental changes may invalidate a well-founded safety argument for different reasons as follows:

1. Changing the argument structure.
2. Evidence is valid only in the operational and environmental context in which it is obtained, or to which it applies. During or after a system change, evidence might no longer support the developers' claims because it could reflect old development artefacts or old assumptions about operation or the operating environment.
3. In the updated system, existing safety claims might be nonsense, no longer reflect operational intent, or they might be contradicted by new data.

The certification process must be repeated after applying changes to an already certified system (i.e., re-certification). In other words, the safety case of the certified system should show that the system is acceptably safe to operate in its intended context after applying the changes. In order to achieve the re-certification, a safety argument should be maintained by determining whether the item of evidence still supports the claims made about it, check whether new or updated safety requirements are reflected in the argument, and review the overall logic of the argument. The main problem though is that the elements of the safety argument (i.e., safety goals, evidence, argument and the operating context) are highly interdependent so that what can be seen as a minor change in the argument may have a major impact to the contents and the structure of that argument [4]. Hence, maintaining a safety argument requires high awareness of the dependencies among its contents and how a change to one part may invalidate other parts. Without this vital awareness, a developer performing impact analysis might not notice that a change has compromised system safety. The Ariane 5 rocket which crashed forty seconds after take-off in 1996 is a costly example of omitting affected parts of a system due to a change. Ariane 5 inertial reference system (SRI) tried to stuff a 64-bit number into a 16-bit space which led to a conversion error. This part of the system was reused from an older version of the SRI that was implemented for Ariane 4 rocket. Seemingly, an assumption was made as since the code was successfully used in an older version of the system then it is suitable to be reused for the newer version [5]. Hence, system developers focused on more complex parts of the system and no attention was paid to the out-of-date code or to any related assumption. A fundamental step prior to update a safety case due to a change is to assess the impact of this change in the safety argument. This is referred to as safety case impact analysis. It is probably clearer now how the continuous maintenance efforts to keep the safety case always up-to-date add more burden on top of the discussed difficulties above. Moreover, the cost of change has become a major part of the cost of ownership of a system [6].

As a response to these challenges, an ambition emerged to modularize safety cases by applying the principles of software architecture and design to the safety case domain. The main idea of the modularity is to align boundaries of safety case modules with design boundaries to contain changes. Having done that, a change to a design element should then affect the corresponding safety case module, and not impact the entire safety argument [6].

To this end, the Industrial Avionics Working Group (IAWG) represented by a team of highly experienced engineers, experts in software development and safety assurance, defined the Modular Software Safety Case (MSSC) process [7] as a means for containing the cost of change by dividing the safety case into a set of argument modules. The process has been refined through experience gained from large-scale trial applications of the prototype process, and further trials of the refined process. MSSC process establishes component traceability mechanism between system design elements and safety argument modules by using the concepts of Dependency-Guarantee Relationship (DGR) and Dependency-Guarantee Contract (DGC). The former is to highlight, and describe, safety-related properties and behaviour of a single design element. In other words, DGRs capture the relationships between input and output ports for each design element. A DGC, however, is used to match one design element's dependencies with another design element's guarantees [8].

The contributions of this paper are as follows: demonstrating how to apply the IAWG MSSC process. More specifically, apply the process to the Fuel Level Estimation System (FLES), which is a real safety critical system that was implemented by Scania AB (a major Swedish automotive industry manufacturer) to show (1) how the DGR and DGC concepts can be used to capture the safety requirements of the FLES, (2) how these two concepts can be used to build a safety case in conformance to the requisites of ISO 26262 for certification, and (3) extending IAWG's DGC to improve the impact analysis process thus facilitating the maintenance of safety cases. This paper is composed of four further sections. In Section 2 we present background information. In Section 3 we present the IAWG MSSC process. In Section 4 we use the FLES to demonstrate the application of the IAWG MSSC process. Finally, in Section 5 we draw conclusions and identify future work.

7.2 Background

This section presents background information about the safety standard ISO 26262, the Goal Structuring Notation (GSN), safety case maintenance and current challenges, and an approach to maintaining safety case evidence after a system change.

7.2.1 The Safety Standard ISO 26262

The rationale behind the selection of this standard for this work is that it is functional safety standard was adapted for automotive electric/electronic systems that Scania is working to qualify for its certification stamp. Since FLES is one of other systems in Scania's trucks, it is very appropriate to consider ISO 26262 for the given example in this paper. ISO 26262 regulates the automotive domain, more specifically, the standard is intended to be applied to safety-related systems that include one or more electrical and/or electronic systems and that are installed in series production passenger cars with a maximum gross vehicle mass up to 3500 kg [9]. In this subsection, however, we focus only on the part of the standard that regulates the decomposition of safety requirements. The following parts are summarized descriptions of the safety requirements decomposition directly from ISO 26262 guidelines:

1. Successively after identifying hazards, the standard recommends to formulate the Safety Goals (SGs) related to the prevention or mitigation of the hazardous events, in order to avoid unreasonable risk. Basically, hazard analysis, risk assessment and Automotive Safety Integrity Level (ASIL) are used to determine the safety goals such that an unreasonable risk is avoided. The standard defines a safety goal as a top-level safety requirement resultant of the hazard analysis and risk assessment. Safety goals are not expressed in terms of technological solutions, but in terms of functional objectives. [9]

2. Identification of safety goals leads to the functional safety concept. The objective of the functional safety concept is to derive the Functional Safety Requirements, from the safety goals, and to allocate them to the preliminary architectural elements. To comply with the safety goals, the functional safety concept contains safety measures, including the safety mechanisms, to be implemented in the item's architectural elements and specified in the functional safety requirements. The standard defines a functional safety requirement as a specification of implementation-independent safety behaviour, or implementation-independent safety measure, including its safety-related attributes. [9]

3. Finally, both the functional concept and the preliminary architectural assumptions lead to the technical safety concept. The first objective of this concept is to specify the Technical Safety Requirements and their allocation to system elements for implementation by the system design. The second objective is to verify through analysis that the technical safety requirements comply with the functional safety requirements. The standard defines a technical safety requirement as a requirement derived for implementation of associated functional safety requirements. [9]

7.2.2 The Goal Structuring Notation (GSN)

A safety argument organizes and communicates a safety case, showing how the items of safety evidence are related and collectively demonstrate that a system is acceptably safe to operate in a particular context. GSN [10] provides a graphical means of communicating (1) safety argument elements, claims (goals), argument logic (strategies), assumptions, context, evidence (solutions), and (2) the relationships between these elements. The principal symbols of the notation are shown in Figure 7.1 (with example instances of each concept).

A goal structure shows how goals are successively broken down into ('solved by') sub-goals until a point is reached where claims can be supported by direct reference to evidence. Using GSN, it is also possible to clarify the argument strategies adopted (i.e., how the premises imply the conclusion), the rationale for the approach (assumptions, justifications) and the context in which goals are stated. It is worth noting that GSN has been extended to enable modularity in a safety case (i.e., module-based development of the safety case). Hence, modular GSN enables the partitioning of a safety case into an interconnected set of modules.

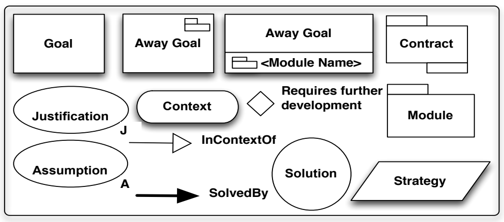


Figure 7.1: Overview of Goal Structuring Notation (GSN)

7.2.3 Safety Case Maintenance and Current Challenges

A safety case is a living document that should be maintained whenever some aspect of the system, its operation, its operating context, or its operational history changes. In this paper, the process of updating the safety case after implementing a system change is referred to as safety case maintenance.

Developers of safety critical systems experience difficulties in safety case maintenance after implementing a system change. One of the main difficulties is identifying the impacted parts in the safety argument. The traceability between a system design and the corresponding safety argument contents, and the dependency among the contents of safety argument are considered two main burdens that encounter the identification of the impacted parts in an argument. Moreover, individual systems tend to become more complex as they are designed and constructed, this increasing complexity, as well as, the number of evidence items in a safety argument can exacerbate the maintenance difficulties. Any approach intends to manage safety argument due to system changes should consider:

1. A means for clearly capturing the underlying rationale of the safety argument in order to assess the impact of change on all parts of the argument.

2. A traceability mechanism between a system domain and the safety argument to support the ability to track the changed part from the system design down to the corresponding affected part in the safety argument.

3. Mechanisms to structure the argument so as to contain the impact of changes.

The use of GSN approach helps to produce well-structured arguments that clearly demonstrate the argument elements and their interdependencies (the relationships between the argument claims and evidence) [11, 12, 4]. Using GSN makes capturing the underlying rationale of the argument easier, which will in turn, help to scope areas affected by a particular change and thus helps the developers to mechanically propagate the change through the goal structure. However, GSN does not tell if the suspect elements of the argument in question are still valid. For example, having made a change to a model we must ask whether goals articulated over that model are still valid. Expert judgment, therefore, is still required in order to answer such questions. Hence, using GSN does not directly help to maintain the argument after a change, but it can more easily determine the questions to be asked to do so [12].

Current standards and analysis techniques assume a top-down development approach to system design. For component-based systems, monolithic evidence produced via these approaches is difficult to maintain those systems because it is hard to match a safety argument that has a different structure than the system design structure. However, safety is a system level property and assuring this property requires every piece of evidence generated for each component to be linked and compared to demonstrate consistency [7]. One may think that the matching (i.e., optimal level of traceability) can be achieved by designing a safety argument structure to be similar to the system design structure, where a clear one-to-one mapping of a system design component to a safety argument module can be established (see Figure 7.2).

Theoretically, a one-to-one mapping may facilitate tracking down the components of a system design to the safety argument, but it is impractical due to four key factors: (1) modularity of evidence, (2) modularity of the system, (3) process demarcation (e.g., ISO 26262 items [9]), and (4) organisational structure (e.g., who is working on what). These factors have a significant influence when deciding upon the safety argument structure.

Enabling component and evidence traceability is very useful to analyse the

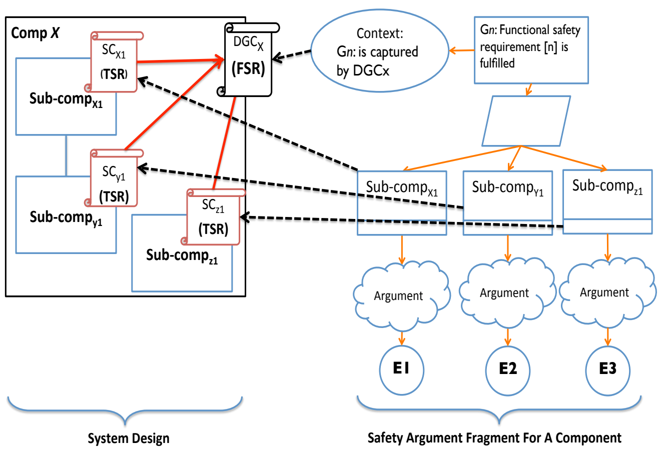


Figure 7.2: An illustration of the relationship between a system design and its safety argument

impact of change on a safety argument, and eventually, facilitates the overall maintenance of the safety case. This paper deals with two forms of traceability: component (i.e. safety argument fragment to system design component) and evidence (i.e. safety argument fragment to supporting evidence). However, to the best of our knowledge there are no supporting process or method that provides detailed steps of how to analyse the impact of a change on a safety case using component or evidence traceability. That said there are well-regarded industry-lead initiatives that assume such methods exist. MSSC Process is one such example.

In this paper, we use the word ‘traceability’ to indicate two different things. Firstly, we refer to the ability to relate safety argument fragments to system design components as component traceability mechanism (through a safety argument). Secondly, we refer to the ability to relate safety argument evidence across system’s artefacts as evidence traceability.

7.2.4 Maintaining Safety Case Evidence after a System Change

In our previous work [1], we proposed a new approach to facilitating safety case change impact analysis. In the approach, automated analysis of information given as annotations to a safety argument (recorded in GSN) highlight suspect safety evidence to bring it to engineer's attention. We proposed annotating each reference to a development artefact (e.g. an architecture specification) in a goal or context element with an artefact version number. We also proposed annotating each solution element with:

1. An evidence version number.
2. An input manifest identifying the inputs (including version) from which the evidence was produced.
3. The lifecycle phase during which the evidence obtained (e.g. Software Architecture Design).
4. A safety standard reference to the clause in the applicable standard (if any) requiring the evidence (and setting out safety integrity level requirements).

With this data, we can perform a number of automated checks to identify items of evidence impacted by a change. For example:

1. We can determine when two different versions of the same item of evidence are cited in the same argument.
2. We can identify out-of-date evidence by searching for input manifests $m = \{(a_1, v_1), ..., (a_n, v_n)\}$ and artefact versions (a, v) such that $\exists i \bullet a = a_i \wedge v > v_i$.
3. Where we know a particular artefact has changed, we can search for input manifests containing old versions.

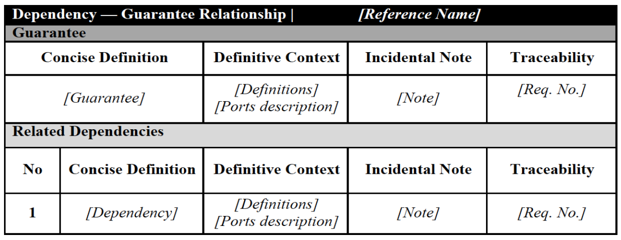
If we had further information which inputs were used to produce each input listed in each input manifest, each input that was used to produce those, and so on, we could extend checks (2) and (3) above to indirect inputs. For example, suppose that life testing is used to establish the reliability of a component, that this component and its reliability appear in a Failure Modes and Effects Analysis (FMEA), and that the FMEA results are used in a Fault Tree Analysis (FTA). With the additional information, we could compute a closure of the FTA's input manifest that would include the life testing results. Other analyses may be possible. For example, we suggest storing the safety standard reference to facilitate analysis of impacts that change the safety integrity level of a requirement.

7.3 Modular Software Safety Case (MSSC) Process

IAWG has proposed Modular Safety Cases as a means of containing the cost of change by dividing the safety case into a set of argument modules. IAWG's MSSC process facilitates handling system changes as a series of relatively small increments rather than occasional major updates (i.e., incremental certification). MSSC process manages system changes by breaking down a system into blocks. The process defines the block as an identifiable part (or group of parts) of the Software implementation that is chosen by the safety case architect to be the subject of a safety case module. Blocks cover all parts of a system design where each block may correspond to a single or multiple software component or unit of design, but it is subject to only one dedicated safety case module. In other words, each system block has one-to-one relationship with a safety argument module. [7]

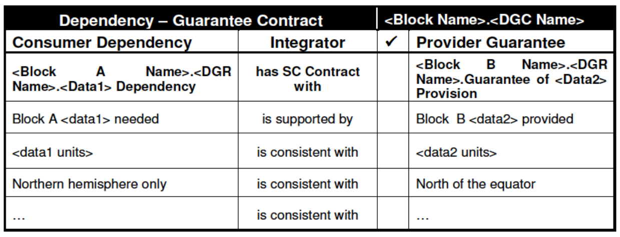
The process establishes component traceability mechanism between system blocks and safety argument modules by using the concepts of DGR and DGC as shown in Figure 7.3 and 7.4, respectively. The former is to highlight and describe safety-related properties and behaviour of a system block. In other words, a DGR captures the relationships between input and output ports for each design block. A DGC, however, is used to match one block's dependencies with another block's guarantees [7, 13]. Creating DGCs leads to the creation of a 'daisy chain' as a dependency in one block and a guarantee offered by another, whose associated dependencies are supported by further guarantees, and so on [13].

MSSC process is very dependent on the anticipated changes that should be identified in the first step of the process. The anticipated change scenarios will bring the highly likely changeable parts in the system to developer's attention.



Dependency — Guarantee Relationship |		*[Reference Name]*		
Guarantee				
Concise Definition		**Definitive Context**	**Incidental Note**	**Traceability**
[Guarantee]		*[Definitions]* *[Ports description]*	*[Note]*	*[Req. No.]*
Related Dependencies				
No	**Concise Definition**	**Definitive Context**	**Incidental Note**	**Traceability**
1	*[Dependency]*	*[Definitions]* *[Ports description]*	*[Note]*	*[Req. No.]*

Figure 7.3: A DGR tabular representation



Dependency – Guarantee Contract			<Block Name>.<DGC Name>
Consumer Dependency	**Integrator**	**✓**	**Provider Guarantee**
<Block A Name>.<DGR Name>.<Data1> Dependency	**has SC Contract with**		**<Block B Name>.<DGR Name>.Guarantee of <Data2> Provision**
Block A <data1> needed	is supported by		Block B <data2> provided
<data1 units>	is consistent with		<data2 units>
Northern hemisphere only	is consistent with		North of the equator
…	is consistent with		…

Figure 7.4: A DGC tabular representation

These scenarios are considered by system developers so that they can manage the containment of the impact of these changes in the system blocks boundaries more efficiently. Having done this, the impact of a change in one safety argument module will hopefully not propagate into another module, but it might impose one (or more) safety case contract update, and even if it is then the cost of changes can be minimised.

It is very important to distinguish between a DGC and a safety case contract. The former captures the required link between a dependency declared in one DGR and a satisfying guarantee provided by another. Hence, DGCs are created on the system design level. A safety case contract, however, is used to describe the linkage between a consumer goal in one Safety Case Module and a provider goal in another [7]. This is formed through the new GSN extension for modularity.

Figure 7.5 shows an example to describe the relationships between system blocks, DGR, DGC, safety case contract and the safety case architecture. It is worth noting that DGCs may be linked to safety case contracts.

The following is a list summarises MSSC process's steps [7]:

Step 1. Analyse the product lifecycle: It is important to predict the potential change scenarios over the projected system lifetime. One reason for that is because change scenarios will help assess the potential benefits that may be achieved through modular certification. If as a result of the analysis there are no changes expected, then the full benefits of modular certification may not be realised, and it may therefore be decided not to adopt a modular approach. [13]

Step 2. Optimise software design and safety case architecture: Since each system block is subject to safety case module. First, we need to divide the system into blocks and form public interfaces for the block

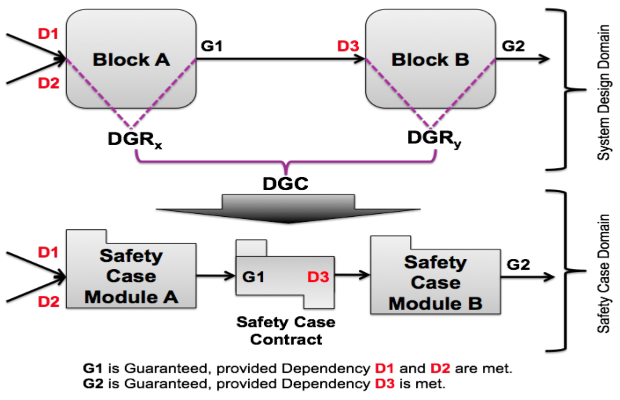


Figure 7.5: Linking blocks using DGRs and DGCs

safety case modules. All elements of the system are split into blocks and each corresponding block safety case module should present an argument about the safety-related behaviour of that block. Second, other necessary modules will be added, for example, software safety requirements, software system wide issues module, configuration data module, safety case contract modules, etc. Finally, we should define safety case integration modules — these provide the argument about the combined behaviour of interdependent safety case modules. [7]

Step 3. Construct safety case modules: A hazard mitigation argument should be formed and derived safety requirements are directed to SW blocks safety case modules. The guaranteed behaviour offered by each block in support of these is captured, along with dependencies on other blocks. A Block Safety Case Module is constructed providing argument and evidence for each Block based on the Guarantees and Dependencies. [7]

Step 4. Integrate safety case modules: The safety case modules are integrated so that claims requiring support in one Safety Case Module are linked to claims providing that support in others. This step of the process results in a fully integrated Safety Case. [7]

Step 5. Assess/Improve change impact: When a system change is implemented, the impact on the design modules and associated Safety Case Modules is assessed. [7]

Step 6. Reconstruct safety case modules

Step 7. Reintegrate safety case modules

Step 8. Appraise the safety case

The guidance of MSSC process [7] does not show detailed information about how to follow some steps including the impact analysis part. The provided example by the process abstracts the impact analysis step and shows its results only. The main work in this paper is not to consider all parts of MSSC process to give a full example on how to apply them but we rather focus on the impact analysis part and necessary prerequisite steps only.

7.4 Illustrative Example: Fuel Level Estimation System (FLES)

In our previous work [14, 15], we used FLES as a specimen system to illustrate the contribution of the architectural model checking to conduct preliminary safety assessment in line with the safety standard ISO 26262.

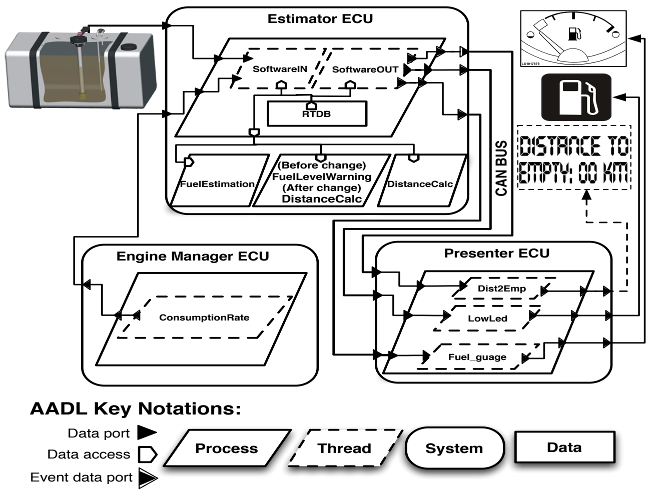


Figure 7.6: An AADL representation of Estimator's software architecture

We used the Architecture Analysis and Design Language (AADL) to model the system as shown in Figure 7.6. In our current work we reuse the description as well as the AADL of FLES to partially apply MSSC process. We also propose a system change scenario and examine how the method helps to highlight the affected safety argument elements.

7.4.1 FLES Description

FLES Technical details

FLES estimates the volume of fuel in a heavy road vehicle's tank and presents this information to the driver through a dashboard mounted fuel gauge. Additionally, the system must warn the driver when this volume falls below a predefined threshold. This system is considered safety critical because its failure could lead to loss of control of the vehicle. For example, if there is less fuel remaining than the driver thinks, the vehicle might run out, bringing it to an unexpected halt, which can be hazardous in certain contexts. As well as bringing the vehicle to a halt, the power steering and braking mechanisms could also fail. These failures would compromise vehicle controllability and could also lead to a crash.

Fuel volume is estimated using a float sensor in the fuel tank. As the position of the float is affected by vehicle motion (negotiating steep hills, sharp bends, or rough terrain), the system has some challenging issues to be tackled within its design. The system must process this signal so that at all times the gauge displays an accurate measurement of the total volume of fuel remaining. The sensed value is sent to the *Estimator ECU*. An Analogue to Digital Converter (ADC) is used to convert and then the *SoftwareIN* thread reads the sensed fuel float position from the ADC and stores it in the real-time database *RTDB*. *FuelEstimation* reads this sensor value and computes an estimate of the current fuel volume in liters. When the vehicle might be moving (i.e., its parking brake is not set), the *FuelEstimation* thread uses a Kalman filter algorithm to reduce the noise introduced by vehicle motion. This algorithm requires the recent history of fuel volume estimates to be stored. *FuelEstimation* outputs a smoothed fuel volume estimate to the *RTDB*. *FuelLevelWarning* then reads this estimate, compares it to the low-fuel warning threshold (i.e., $< 7\%$ of the fuel tank capacity), and writes the low-fuel warning status to the *RTDB*. *SoftwareOUT* reads the fuel volume and low-fuel warning status from the *RTDB* and sends these over the Controller Area Network (CAN) bus to the *Presenter ECU*. The *Presenter ECU* adjusts the actuators (i.e., fuel gauge and low-fuel lamp) on the dashboard according to the received values.

FLES safety analysis

Hazard analysis and risk assessment made for FLES led to one hazard identification: *"Unannunciated lack of fuel"*. Unannunciated is interpreted as (1) fuel estimates and low-fuel warning are not displayed at all, and (2) it is displayed incorrectly since the estimates are not identical to the real amount of fuel in the vehicle's tank. The determined ASIL for the fuel level estimation system is "C". The derived safety requirements to mitigate the hazard are decomposed as recommended by ISO 26262 as follows:

1. ***Safety goals:*** Two safety goals were derived:
   (a) *SG1.0ImplAssur*: Vehicle's driver shall be constantly aware of the actual remaining fuel in the tank whenever the engine is in operation.
   (b) *SG2.0ImplAssur*: Vehicle's driver shall be warned when the fuel level is low and the engine is in operation.
2. ***Functional Safety Requirements (FSR):***
   - Two functional safety requirements were identified to satisfy *SG1.0ImplAssur*:
     (a) *ConFSR1.0.1.0*: A fuel gauge should promptly annunciate the actual fuel amount in the tank whenever the engine is in operation.
     (b) *ConFSR1.0.2.0*: The fuel gauge shall not display a fuel estimate that deviates more that 5% from the actual fuel volume in the tank.
   - One functional safety requirement was identified to satisfy *SG2.0ImplAssur*:
     (c) *ConFSR2.0.1.0*: A fuel-low warning lamp should be promptly turned ON when the fuel level in the tank falls below a certain level whenever the engine is in operation.
3. ***Technical Safety Requirements (TSR):*** There is a large set of technical safety requirements that was identified to specify the functional safety requirements. The work of the paper, however, considers the minimum set of technical safety requirements that specify *ConFSR1.0.1.0* and *ConFSR2.0.1.0* as shown in Table 7.1.

Table 7.1: A Subset of the identified TSRs for FLES

FSR ID	TSR ID	Description
FSR1.0.1.0	F1010TSR1	The *FuelEstimation* thread shall provide the *totalFuelLevel* value
FSR1.0.1.0	F1010TSR2	The *SoftwareOUT* shall send the *totalFuelLevel* value to the *Presenter*
FSR2.0.1.0	F2010TSR1	The *FuelLevelWarning* thread shall value provide *lowFuelWarning*
FSR2.0.1.0	F2010TSR2	The *SoftwareOUT* shall send the *lowFuelWarning* value to the *Presenter*

7.4.2 Applying the IAWG MSSC Process

A list of anticipated change scenarios during FLES's lifetime is required. This list may help assessing the potential benefits that may be achieved through modular certification. In this section, we present the details of the various MSSC process steps with respect to FLES:

Analyse the product lifecycle and identify change scenarios

We assume one potential change for FLES. The *Distance To Empty* feature might be added to FLES. The role of this anticipated change is to determine the distance (Km) that a vehicle can drive before it runs out of fuel. This new feature is dependent on (1) the estimation of the current fuel amount in the tank (L), and (2) the fuel consumption rate (L/Km) in the engine. Technically, this intended feature will be added as a new thread in the *Estimator ECU*. This thread should read the output of the *FuelEstimation* thread, as well as, the output of the *ConsumptionRate* thread that is implemented in the *EngineManager ECU*. To avoid dealing with timing and memory budgets, FLES engineers expect to remove the *FuelLevelWarning* thread and move the task it contains to the *FuelEstimation* thread (i.e., merge the two threads into one). Since the safety margin of the *FuelEstimation* thread allows adding a new task, the timing and memory budget for the thread will remain the same even after the merge. On the other hand, the new *DistanceCalc* thread will take the timing and memory budget, and the priority of the removed *FuelLevelWarning* thread. The same arrangements will be applied to the threads in the *Presenter ECU*.

Optimise software design and safety case architecture (define the safety case architecture)

For the sake of simplicity, we do not define a full set of the safety case modules, but we rather define the basic modules that are sufficient to make the example. We focus on the *Estimator ECU* in our example by dividing it into two software blocks, namely, *FuelEstimationBK* and *FuelLevelWarningBK*. Each of them represents a safety case module. Additionally, we construct *Hazard Mitigation*, *SW Safety Requirements* and *SW Integration test* modules (as shown in Figure 7.7).

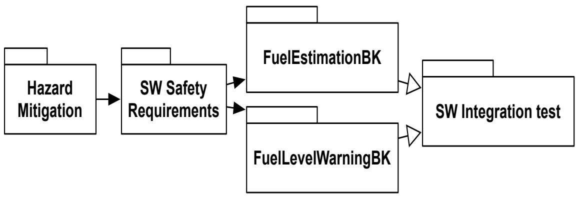


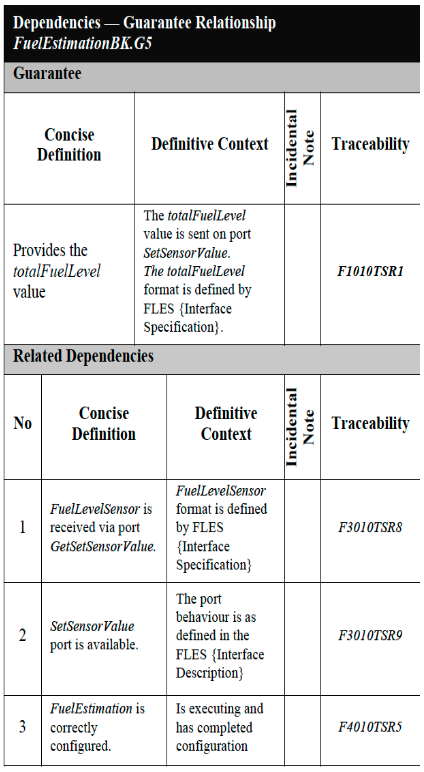
Figure 7.7: FLES safety case architecture

Construct safety case modules and Integrate safety case modules

We merge these two steps for the sake of simplicity. We identify the required DGRs of the *FuelEstimationBK* and *FuelLevelWarningBK* blocks. We also construct the *Hazard Mitigation*, *SW Safety requirements*, *FuelEstimationBK*, *FuelLevelWarningBK*, and *Software Integration test* safety case modules.

Table 7.2 shows one DGR for the software block *FuelEstimationBK* in which the block (i.e., represented as thread) guarantees that it can provide the estimated fuel level volume in the tank *totalFuelLevel* if the three dependencies are met. Table 7.3 shows one DGR for the software block *FuelLevelWarningBK* in which the block (i.e., represented as thread) guarantees that it can tell if the fuel is low or not (*lowFuelLevelWarning* is True if the fuel is below 7% of the tank capacity and False if the fuel is not) once the four related dependencies are met.

In Figure 7.8, we construct the hazard mitigation argument. Basically, *MitigationHazard1* goal is supported by implementing and assuring the two safety goals that were derived to mitigate it. The safety goals are represented by the two separated away goals *SG1.0ImplAssur*, and *SG2.0ImplAssur*. These goals



Dependencies — Guarantee Relationship
FuelEstimationBK.G5

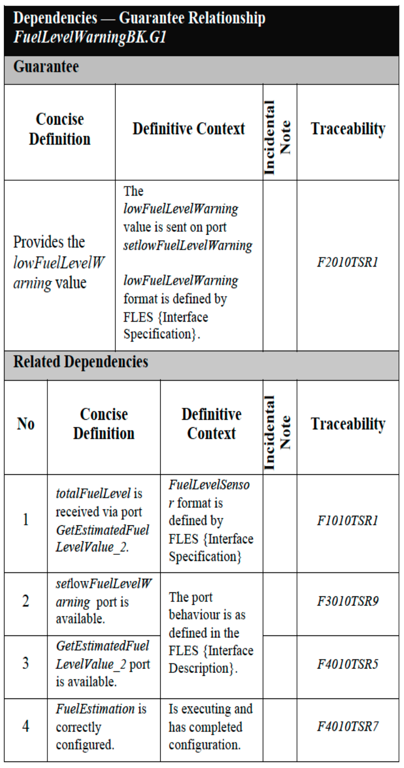
Guarantee

Concise Definition	Definitive Context	Incidental Note	Traceability
Provides the *totalFuelLevel* value	The *totalFuelLevel* value is sent on port *SetSensorValue*. *The totalFuelLevel* format is defined by FLES {Interface Specification}.		*F1010TSR1*

Related Dependencies

No	Concise Definition	Definitive Context	Incidental Note	Traceability
1	*FuelLevelSensor* is received via port *GetSetSensorValue*.	*FuelLevelSensor* format is defined by FLES {Interface Specification}		*F3010TSR8*
2	*SetSensorValue* port is available.	The port behaviour is as defined in the FLES {Interface Description}		*F3010TSR9*
3	*FuelEstimation* is correctly configured.	Is executing and has completed configuration		*F4010TSR5*

Table 7.2: DGR FuelEstimationBK



Dependencies — Guarantee Relationship
FuelLevelWarningBK.G1

Guarantee

Concise Definition	Definitive Context	Incidental Note	Traceability
Provides the *lowFuelLevelWarning* value	The *lowFuelLevelWarning* value is sent on port *setlowFuelLevelWarning* *lowFuelLevelWarning* format is defined by FLES {Interface Specification}.		*F2010TSR1*

Related Dependencies

No	Concise Definition	Definitive Context	Incidental Note	Traceability
1	*totalFuelLevel* is received via port *GetEstimatedFuelLevelValue_2*.	*FuelLevelSensor* format is defined by FLES {Interface Specification}		*F1010TSR1*
2	*setlowFuelLevelWarning* port is available.	The port behaviour is as defined in the FLES {Interface Description}.		*F3010TSR9*
3	*GetEstimatedFuelLevelValue_2* port is available.			*F4010TSR5*
4	*FuelEstimation* is correctly configured.	Is executing and has completed configuration.		*F4010TSR7*

Table 7.3: DGR FuelLevelWarningBK

also represent the integration between *Hazard Mitigation* and *SW Safety Requirements* safety case modules (see Figure 7.9).

In *FuelLevelWarning.BK* Safety case module (see Figure 7.10), we show how arguing over the dependencies supports the guarantee that is represented by *FuelLevelWarningBK.G1*. The argument module uses *FuelEstimationBK.G5* as a dependency to support the guarantee. *FuelEstimationBK.G5* also relies on a set of dependencies to be guaranteed. Figure 7.11 shows an argument fragment of the *SW Integration test* safety case module. The objective of the module is to argue over the integration of the software elements within the *Estimator ECU*.

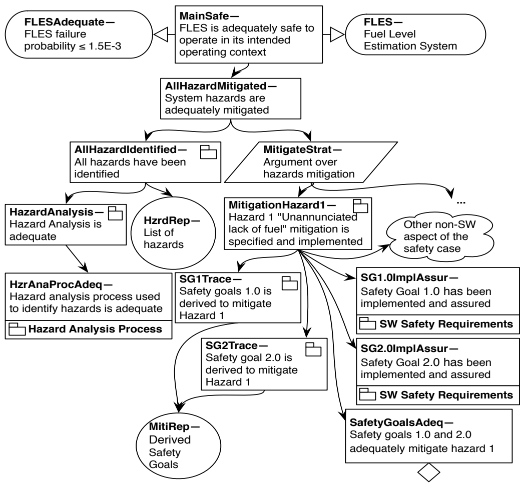

FLESAdequate—
FLES failure probability ≤ 1.5E-3
MainSafe—
FLES is adequately safe to operate in its intended operating context
FLES—
Fuel Level Estimation System
AllHazardMitigated—
System hazards are adequately mitigated
AllHazardIdentified—
All hazards have been identified
MitigateStrat—
Argument over hazards mitigation
...
HazardAnalysis—
Hazard Analysis is adequate
HzrdRep—
List of hazards
MitigationHazard1—
Hazard 1 "Unannunciated lack of fuel" mitigation is specified and implemented
Other non-SW aspect of the safety case
HzrAnaProcAdeq—
Hazard analysis process used to identify hazards is adequate
Hazard Analysis Process
SG1Trace—
Safety goals 1.0 is derived to mitigate Hazard 1
SG1.0ImplAssur—
Safety Goal 1.0 has been implemented and assured
SW Safety Requirements
SG2Trace—
Safety goal 2.0 is derived to mitigate Hazard 1
SG2.0ImplAssur—
Safety Goal 2.0 has been implemented and assured
SW Safety Requirements
MitiRep—
Derived Safety Goals
SafetyGoalsAdeq—
Safety goals 1.0 and 2.0 adequately mitigate hazard 1

Figure 7.8: Hazard mitigation safety case module of FLES

The *FuelLevelWarningBK.G1* DGR shows that in order for *FuelLevelWarningBK* being able to fulfill the *TSR F2010TSR1* it requires the *TSR F1010TSR1*, which is guaranteed by a different DGR (i.e., *FuelEstimationBK.G5*). Here lies the importance of the DGC as it matches such dependencies. Table 7.4 shows a DGC that matches *F2010TSR1* to *F1010TSR1*. MSSC process requires performing the integration of safety case modules by using a safety case contract module. The latter uses a DGC to set out the matching between the DGRs of the goals involved. However, since our work is more focused on facilitating the impact analysis within the blocks, we do not use safety case contracts in this example thus no goals are supported by contracts. The integration, in our example, is done through public and away goals.

Assess/Improve change impact

In this step, we use our approach for maintaining safety cases (Section 7.2.4) to extend IAWG's DGC. We use the extended DGC in the FLES example to show how the extension can help: (1) highlighting the affected argument elements, and (2) identifying inadequacies in the generated artefacts from the development lifecycle of FLES.

Table 7.4 shows an extended DGC of *FuelLevelWarning.BK*. The exten-

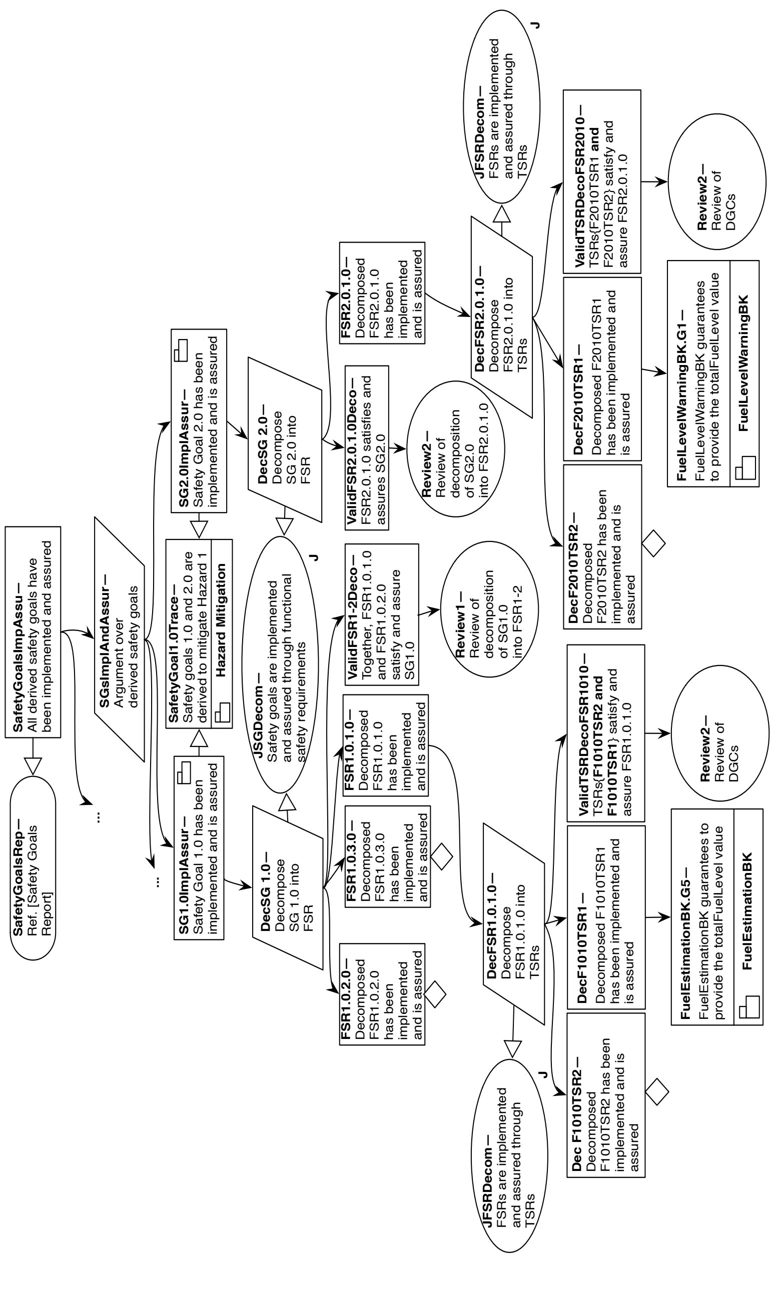

SafetyGoalsImpAssu— All derived safety goals have been implemented and assured
SafetyGoalsRep— Ref. [Safety Goals Report]
SGsImplAndAssur— Argument over derived safety goals
...
SafetyGoal1.0Trace— Safety goals 1.0 and 2.0 are derived to mitigate Hazard 1
Hazard Mitigation
SG2.0ImplAssur— Safety Goal 2.0 has been implemented and is assured
SG1.0ImplAssur— Safety Goal 1.0 has been implemented and is assured
...
DecSG 2.0— Decompose SG 2.0 into FSR
JSGDecom— Safety goals are implemented and assured through functional safety requirements
J
DecSG 1.0— Decompose SG 1.0 into FSR
FSR2.0.1.0— Decomposed FSR2.0.1.0 has been implemented and is assured
ValidFSR2.0.1.0Deco— FSR2.0.1.0 satisfies and assures SG2.0
ValidFSR1-2Deco— Together, FSR1.0.1.0 and FSR1.0.2.0 satisfy and assure SG1.0
FSR1.0.1.0— Decomposed FSR1.0.1.0 has been implemented and is assured
FSR1.0.3.0— Decomposed FSR1.0.3.0 has been implemented and is assured
FSR1.0.2.0— Decomposed FSR1.0.2.0 has been implemented and is assured
Review2— Review of decomposition of SG2.0 into FSR2.0.1.0
Review1— Review of decomposition of SG1.0 into FSR1-2
DecFSR2.0.1.0— Decompose FSR2.0.1.0 into TSRs
JFSRDecom— FSRs are implemented and assured through TSRs
J
DecFSR1.0.1.0— Decompose FSR1.0.1.0 into TSRs
JFSRDecom— FSRs are implemented and assured through TSRs
J
ValidTSRDecoFSR2010— TSRs(F2010TSR1 and F2010TSR2) satisfy and assure FSR2.0.1.0
DecF2010TSR1— Decomposed F2010TSR1 has been implemented and is assured
DecF2010TSR2— Decomposed F2010TSR2 has been implemented and is assured
ValidTSRDecoFSR1010— TSRs(F1010TSR2 and F1010TSR1) satisfy and assure FSR1.0.1.0
DecF1010TSR1— Decomposed F1010TSR1 has been implemented and is assured
Dec F1010TSR2— Decomposed F1010TSR2 has been implemented and is assured
Review2— Review of DGCs
FuelLevelWarningBK.G1— FuelLevelWarningBK guarantees to provide the totalFuelLevel value
FuelLevelWarningBK
Review2— Review of DGCs
FuelEstimationBK.G5— FuelEstimationBK guarantees to provide the totalFuelLevel value
FuelEstimationBK

Figure 7.9: SW Safety Requirements safety case module

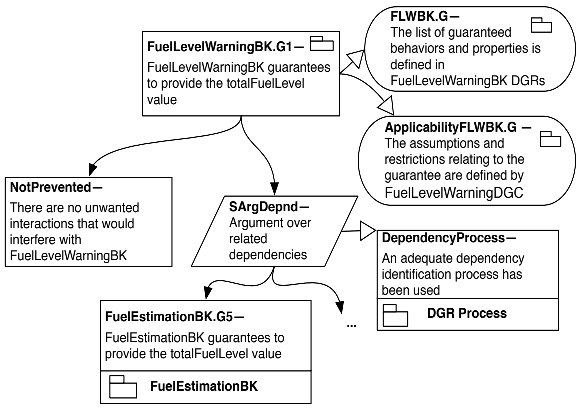

FuelLevelWarningBK.G1—
FuelLevelWarningBK guarantees to provide the totalFuelLevel value
FLWBK.G—
The list of guaranteed behaviors and properties is defined in FuelLevelWarningBK DGRs
ApplicabilityFLWBK.G —
The assumptions and restrictions relating to the guarantee are defined by FuelLevelWarningDGC
NotPrevented—
There are no unwanted interactions that would interfere with FuelLevelWarningBK
SArgDepnd—
Argument over related dependencies
DependencyProcess—
An adequate dependency identification process has been used
DGR Process
FuelEstimationBK.G5—
FuelEstimationBK guarantees to provide the totalFuelLevel value
FuelEstimationBK
...

Figure 7.10: An argument fragment of FuelLevelWarning.BK module

sion is represented by the cells in grey. Moreover, Figure 7.11 shows items of evidence (i.e., GSN solution) that support claims about the consistency among the ports of FLES blocks. The green elements in the figure represent the annotations described in Section 7.2.4.

Table 7.4: FuelLevelWarningDGC

Dependency — Guarantee Contract | FuelLevelWarningDGC			
Consumer Dependency	**Integrator**	**Provider Guarantee**	Artefact Version
FuelLevelWarningBK.G1	Supported by away goal *FuelEstimationBK.G5*	***FuelEstimationBK.G5***	V.3.2
totalFuelLevel value is received	Is Supported By	Provides the *totalFuelLevel* value	
totalFuelLevel value is received via *GetEstimatedFuelLevelValue_2 port*	Is Consistent with	The *totalFuelLevel* value is sent on port *SetSensorValue*.	InConChk TstInnInt
totalFuelLevel data format is defined by FLES {Interface Specification Ref.20}	Is Consistent with	*totalFuelLevel* data format is defined by FLES {Interface Specification Ref.20}	

Supporting Evidence					
No	**GSN element**	**Evidence Version**	**Input Manifest**	**Lifecycle Phase**	**Safety Standard Reference**
1	InConChk	V.3.2	(Inchecker, 1.5), (Code, 1.0)	SW Dev.	§ 8.4.2.2.4 ASIL "C"
2	TstInnInt	V.3.2	(Con1, 3.0), (Code, 3.2)	SW Dev.	§ 8.4.2.2.4 ASIL "C"

Table 7.5 shows the impacted elements of the safety case with a brief explanation for each element.

Table 7.5: Results of change impact analysis

No.	Module Name	Element affected	Explanation
1	*SW Safety Requirements*	DecF1010TSR1	The decomposition of this requirement has been changed
2	*SW Safety Requirements*	DecF2010TSR1	The decomposition of this requirement has been changed
3	*FuelLevelWarning.BK*	The entire module	Merged with another Module
4	*SW Integration test*	EstimaInnInter and all claims below	Argument about the estimation internal interfaces is suspect
5	*SW Integration test*	*InConChk*	Out of date implementation
6	*SW Integration test*	*TstInnInt*	Out of date implementation

Now, let us consider the potential change scenario in Section 7.4.2 to illustrate how the information contained within the annotations aids the change impact analysis in safety arguments. Merging *FuelEstimation* and *FuelLevelWarning* into one thread will impact the consistency of the interfaces and connections of FLES. Suppose that an engineer making this change had updated the artefact version annotation(s) in part of the argument that refers to the interfaces of those threads. An automated implementation of the described checks in Section 7.2.4 could highlight the need to re-run the interface consistency check, as well as, the *Estimator* internal interfaces testing. If the new version of the implementation is version 3.3, analysis of the manifest associated with *InConChk* and *TstInnInt* would reveal evidence based on an older version of the implementation and tools could flag *InConChk* and *TstInnInt* as out-of-date and suspect. Automated analysis might also highlight goal *EstimatorImpCorr* because its artefact version annotation refers to an out-of-date version of the *Estimator* implementation. The goal and its supporting argument are suspect because they might refer to parts of the implementation that no longer exist or make claims about the implementation that are no longer true.

The principal difference between our work and the existing approach pro-

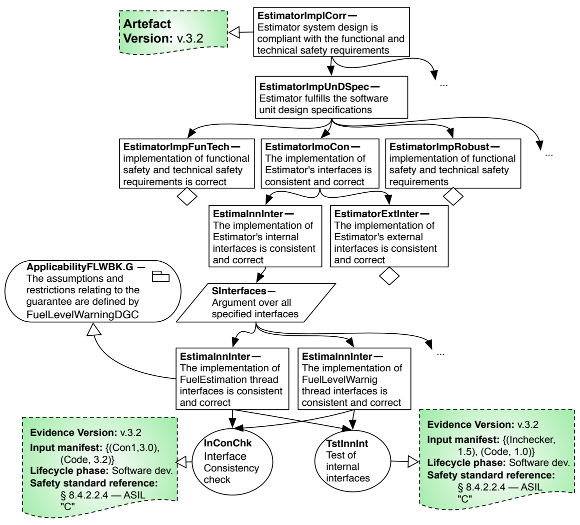

Artefact Version: v.3.2
EstimatorImplCorr — Estimator system design is compliant with the functional and technical safety requirements
...
EstimatorImpUnDSpec — Estimator fulfills the software unit design specifications
EstimatorImpFunTech — implementation of functional safety and technical safety requirements is correct
EstimatorImoCon — The implementation of Estimator's interfaces is consistent and correct
EstimatorImpRobust — implementation of functional safety and technical safety requirements
...
EstimaInnInter — The implementation of Estimator's internal interfaces is consistent and correct
EstimatorExtInter — The implementation of Estimator's external interfaces is consistent and correct
ApplicabilityFLWBK.G — The assumptions and restrictions relating to the guarantee are defined by FuelLevelWarningDGC
SInterfaces — Argument over all specified interfaces
EstimaInnInter — The implementation of FuelEstimation thread interfaces is consistent and correct
EstimaInnInter — The implementation of FuelLevelWarnig thread interfaces is consistent and correct
...
Evidence Version: v.3.2
Input manifest: {(Con1,3.0), (Code, 3.2)}
Lifecycle phase: Software dev.
Safety standard reference: § 8.4.2.2.4 — ASIL "C"
InConChk Interface Consistency check
TstInnInt Test of internal interfaces
Evidence Version: v.3.2
Input manifest: {(Inchecker, 1.5), (Code, 1.0)}
Lifecycle phase: Software dev.
Safety standard reference: § 8.4.2.2.4 — ASIL "C"

Figure 7.11: An argument fragment of SW Integration test safety case module

posed by the IAWG MSSC is that the MSSC approach contains changes at the level of a safety argument module and the corresponding system blocks. In contrast, our approach provides the engineer to contain the changes at a lower-level where they feel that a tighter control over change is needed. More specifically, our approach means that changes can be contained within a safety argument module and within specific system blocks. It could be argued that this could have been handled in the existing approach by decomposing the system and its safety argument differently, however in practice it is better not to constrain system architects unnecessarily.

7.5 Conclusion and Future Work

Applying changes to systems during their lifetime is inevitable task. In safety critical systems, system changes can be accompanied with changes to safety arguments. Maintaining those arguments is painstaking process because of the dependencies between their elements. The IAWG MSSC process was introduced as a response to safety cases maintenance difficulties. The process recommends applying changes as a series of relatively small increments rather than occasional major ones. However, the guidance of MSSC process does not show detailed information about how to follow some steps including the impact analysis part. In this paper, we applied the process to a real life safety critical system to show how system engineers can identify the elements in a safety argument that might be impacted by a change. We showed that by extending the proposed DGC by IAWG to include additional information as annotations that is useful to highlight the impacted argument elements. Moreover, we provided starting points to maintain the affected parts of the argument as we described the reasons why they have become inadequate due to the change. The impact check based on the additional information is still manual as we have not yet studied the feasibility or value of developing a tool to automate the checks but we leave this effort to future work.

Acknowledgement

We acknowledge the Swedish Foundation for Strategic Research (SSF) SYNOPSIS Project for supporting this work.

Bibliography

[1] Omar Jaradat, Patrick J. Graydon, and Iain Bate. An approach to maintaining safety case evidence after a system change. In *proceedings of the 10th European Dependable Computing Conference (EDCC)*, August 2014.

[2] Preliminary safety case for airborne traffic situational awareness for enhanced visual separation on approach psc atsa-vsa. Technical report, European Organisation for the Safety of Air Navigation, EUROCONTROL, 2011.

[3] Ewen Denney, Ganesh Pai, and Iain Whiteside. Hierarchical safety cases. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *NASA Formal Methods*, volume 7871 of *Lecture Notes in Computer Science*, pages 478–483. Springer Berlin Heidelberg, 2013.

[4] T.P. Kelly and J.A. McDermid. A systematic approach to safety case maintenance. In *proceedings of the Computer Safety, Reliability and Security*, volume 1698 of *Lecture Notes in Computer Science*, pages 13–26. Springer Berlin Heidelberg, 1999.

[5] P. Conmy. *Safety Analysis of Computer Resource Management Software (CRMS).* PhD thesis, University of York, 2007.

[6] Tim Kelly. Modular certification, 2007.

[7] Modular software safety case (MSSC) — process description. https://www.amsderisc.com/related-programmes/, Nov 2012.

[8] Tim Kelly. Using software architecture techniques to support the modular certification of safety-critical systems. In *Proceedings of the Eleventh*

Australian Workshop on Safety Critical Systems and Software - Volume 69, SCS '06, pages 53–65, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[9] ISO 26262:2011. Road Vehicles — Functional Safety, Part 1-9. International Organization for Standardization, Nov 2011.

[10] GSN Community Standard: (http://www.goalstructuringnotation.info/) Version 1; (c) 2011 Origin Consulting (York) Limited.

[11] T. Kelly. Literature survey for work on evolvable safety cases. Department of Computer Sceince, University of York, 1st Year Qualifying Dissertation, 1995.

[12] S P Wilson, T P Kelly, and J A McDermid. Safety case development: Current practice, future prospects. In *proceedings of the 12th Annual CSR Workshop - Software Bases Systems*. Springer-Verlag, 1997.

[13] Jane L Fenn, Richard D Hawkins, PJ Williams, Tim P Kelly, Michael G Banner, and Y Oakshott. The who, where, how, why and when of modular and incremental certification. In *proceedings of the 2nd IET International Conference on System Safety*, pages 135–140. IET, 2007.

[14] Omar Jaradat, Patrick Graydon, and Iain Bate. The role of architectural model checking in conducting preliminary safety assessment. In *proceedings of the 31st International System Safety Conference (ISSC), System Safety Society*, August 2013.

[15] Omar Jaradat. Automated architecture-based verification of safety-critical systems. Master's thesis, Mälardalen University, www.diva-portal.org/smash/record.jsf?pid=diva220 February 2015, 2012.

Chapter 8

Paper C: Deriving Safety Contracts to Support Architecture Design of Safety Critical Systems

Irfan Šljivo, Omar Jaradat, Iain Bate, Patrick Graydon.
In Proceedings of the 16th IEEE International Symposium on High Assurance Systems Engineering (HASE 2015), IEEE, USA, January 2015.

Abstract

The use of contracts to enhance the maintainability of safety-critical systems has received a significant amount of research effort in recent years. However some key issues have been identified: the difficulty in dealing with the wide range of properties of systems and deriving contracts to capture those properties; and the challenge of dealing with the inevitable incompleteness of the contracts. In this paper, we explore how the derivation of contracts can be performed based on the results of failure analysis. We use the concept of safety kernels to alleviate the issues. Firstly the safety kernel means that the properties of the system that we may wish to manage can be dealt with at a more abstract level, reducing the challenges of representation and completeness of the "safety" contracts. Secondly the set of safety contracts is reduced so it is possible to reason about their satisfaction in a more rigorous manner.

8.1 Introduction

Contract-based approaches aimed at decreasing certification costs and increasing maintainability of safety-critical systems have been the topic of much research recently. Many works focus on the underlying contract theory [1, 2, 3], while not that many focus on the difficulty of specifying contracts and the problem of their (in)completeness [4, 5]. A component contract is usually defined as a pair of assumption/guarantee assertions such that the component offers the guarantee if an environment in which the component is used satisfies the assumptions. The contracts can be characterised as either strong or weak [6]. The strong contracts capture behaviours that should hold in all environments/contexts in which the component can be used, while the weak contracts capture context specific behaviours. A "safety contract" is a contract that specifically deals with behaviours of the system linked to hazard mitigation.

Developers of safety-critical systems are sometimes required to construct a safety case to show that the system is acceptably safe to operate in a given context, i.e., that the risks of hazards occurring are reduced to acceptable levels. As a way of documenting the safety case, a safety argument is often used to show how safety claims about the system are connected and supported by evidence. While the argument presents the safety-relevant information about the system in a comprehensible way, safety contracts capture the safety-relevant information in a more rigorous manner. The fact that both, the safety argument and safety contracts, deal with the same information makes the contracts an important aid in safety case maintenance [7].

As safety-critical systems are characterised by a wide-range of properties that influence safety-relevant behaviour of components, it is challenging to derive contracts with a complete set of relevant assumptions on the environment that imply the guaranteed component behaviour. When dealing with completeness of contracts without a reference point against which we can check if the contracts are complete, then the contracts are inevitably incomplete, since we cannot capture all assumptions. To talk about contract completeness we need to identify the reference point against which we can check the contracts and that we can use to derive the contracts as well. For example, safety contracts describing failure behaviours of a component can be derived from a failure analysis such as Fault Tree Analysis (FTA).

Not all failure behaviours obtained by failure analysis are relevant from the perspective of hazard analysis results. Regardless of that, we still categorise contracts capturing such behaviours as *safety* contracts, since the captured behaviours can be safety-relevant in case of change to the system or for other systems in which the component can be used. An approach to developing systems based on a "safety kernel" was first proposed by Rushby [8] and used by Wika [9]. The basic principles of their work are that:

1. The safety kernel protects the system from key (higher criticality) hazardous events by checking that data flowing out of a module of the system would not violate Derived Safety Requirements (DSR) obtained via hazard analysis.

2. The safety kernel itself is much simpler than the rest of the system.

The simplicity means that the safety kernel can be developed to the requisite high integrity even if the rest of the system cannot be. Overall, the system is at least as safe as without a safety kernel but costs may be reduced. In this paper, we extend the original concept to include safety contracts being associated with the safety kernel which to help facilitate incremental certification. The simplicity of the safety kernel also means the aforementioned problems of representing contracts and achieving completeness are eased.

Potential system changes during the system lifetime may impact some parts of the safety case. These affected parts necessitate updating the safety case with respect to those changes. We refer to the updating of the safety case after implementing a system change as *incremental certification*. The intention that change impact analysis can be performed by mainly assessing whether the contracts still hold is slightly unrealistic as there are significant issues with achieving complete contracts [5]. We deem that change impact analysis can be guided by accessing the satisfaction and completeness of contracts with respect to failure analyses.

In this paper, we focus on the safety contract derivation and the issue of their (in)completeness, as these two steps form the basis for establishing safety case maintenance techniques using the safety contracts. We judge that the contract completeness can be established only with respect to a clearly identified reference point such as failure analysis. Since failure analysis itself can be incomplete, the derived safety contracts are at least as complete as the analysis itself. Although contract completeness cannot be established in general, contracts can be used for guiding the designer to the key properties of the system as part of de-risking incremental certification and making it more efficient. This is supplemented by the designer being given scenario-specific guidance on how to deal with certain likely changes. In general for safety-critical systems there is often a clear development roadmap that makes this form of guidance practical. For instance it may be known that in *N* years time that the developers will want to change the processors used due to obsolescence or remove a hydro-mechanical backup due to weight. Maintainers updating or upgrading a system might benefit from the original designers' insight on planned change scenarios[7].

The contribution and structure of the paper is as follows. In section 8.2, we present the related work and an illustrative example used to demonstrate the approach. An architecture and supporting development process, in section 8.3, that allows two types of contracts to be supported that should lead to a reduction in the initial certification costs as well as making the system easier to maintain. In section 8.4, we demonstrate an approach to deriving safety contracts from FTA and present how the derived contracts completeness check could be performed with respect to the fault trees. In section 8.5, we present a safety argument based on the use of the safety kernel and contracts. Finally, we present summary and conclusions in section 8.6.

8.2 Background and Motivation

In this section we present the state of the art related to contracts and modular safety arguments. In the second part of the section we provide a brief description of the computer assisted braking system used to illustrate the approach.

8.2.1 Related Work

We group the related work into two areas: contract-based approaches for safety-critical systems and approaches related to safety arguments for incremental certification.

Use of Contracts in Safety-Critical Systems

An "informal" contract-based approach is proposed in [4]. The approach uses dependency-guarantee relationships to capture dependencies between modules. The captured dependencies are identified by considering predicted changes in the system in order to best contain their impact. A difficulty that arises is that usually not all dependencies can be captured if contracts are restricted to the relationship between just two modules as dependency chains can span across several modules. Furthermore, the issue of contract incompleteness is not fully addressed.

A more formal contract-based approach is shown in [10]. The work presents a language for describing assumption/guarantee contracts used to capture vertical dependencies between a software application and a hardware platform. While the approach provides a benefit of automatic generation of parts of arguments, it does not support capturing the broad range of assumptions needed for a guarantee to be still valid when a change in the environment occurs.

A range of formal contract-based approaches based on contract algebra can be found in [1, 2, 3]. The contract algebra includes definitions of contract refinement, composition, conjunction etc., making these approaches quite powerful when it comes to contract verification. The contract examples provided in [2, 3] do not focus on failure behaviour, but rather on behaviour when no failures occur. Moreover, the presented contracts on timing behaviour require additional assumptions if they are to be used in the process of incremental or modular certification [5]. In our work we propose that in addition to contracts describing expected behaviour in a specific context captured within weak contracts, we capture strong contracts describing how the faults in the system are handled by the safety kernel. Due to the properties of the safety kernel, such contracts are generally easier to satisfy due to fewer assumptions.

Safety Argumentation in Support of Incremental Certification

In safety critical systems, particularly those for which a safety case should be provided, change management is a painstaking process. That is because accommodating the changes in the system domain should be followed by updating the safety case (i.e., incremental certification) in a safe and efficient manner. A process is proposed in [7] to facilitate the incremental change and evolving system capability. One objective of the Modular Software Safety Case (MSSC) process is to minimise the impact on the safety case of changes which might be expected during the life of the system. Using the process may increase the system flexibility to accept changes.

The structure of the argument has a significant role in accommodating the changes. Well structured arguments clearly demonstrate the relationships between the argument claims and evidence, therefore it is easier to understand the impact of changes on them than poorly structured arguments. Moreover, well structured arguments can be exploited to prioritise the handling of change, identify the key areas of concern, and hence de-risk the change management process. An approach is proposed in [11] to show how the safety argument structure facilitates the systematic impact assessment of the safety case after applying changes. More specifically, the proposed approach shows how it

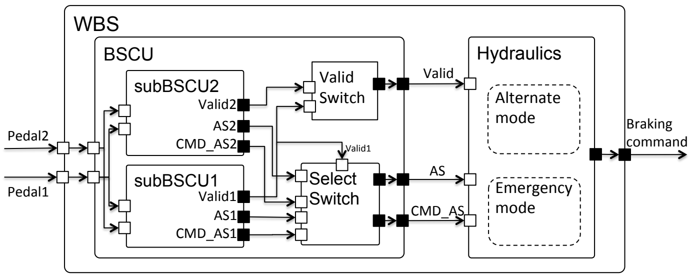


Figure 8.1: Wheel Braking System - High Level View

is possible to use the recorded dependencies of the goal structure to follow through the impact of a change and recover from change.

Another approach is proposed in [12] to facilitating safety case change impact analysis. In this approach, automated analysis of information given as annotations to the safety argument highlights suspect safety evidence that may need updating following a change to the system being performed.

8.2.2 Overview of the Computer Assisted Braking System

In this section we will present the computer assisted braking system of an aircraft used in ARP4761 standard [13] to demonstrate the safety assessment process. The standard describes a Wheel Braking System (*WBS*) that takes two input brake pedal signals and outputs the braking command signal. The high level architecture is shown in Fig. 8.1. For the purposes of this paper we consider that all six components of the WBS shown in Fig. 8.1 are implemented in software.

The system is composed of two subsystems: Brake System Control Unit (*BSCU*) and *Hydraulics*. The brake pedal signals are forwarded to *BSCU*, which generates braking commands and sends the commands via direct link to *Hydraulics* subsystem that executes the braking commands. If the *BSCU*, which makes the normal operation mode, fails then *Hydraulics* uses an alternate mode to perform the braking. If both, normal and alternate mode fail, emergency brake is used.

In order to address the availability and integrity requirements, *BSCU* is designed with two redundant dual channel systems: *subBSCU1* and *subBSCU2*. Each of these subsystems consists of *Monitor* and *Command* components. *Monitor* and *Command* take the same pedal position inputs, and both calculate the command value. The two values are compared within the *Monitor* component and the result of the comparison is forwarded as true or false through *Valid* signal. The *SelectSwitch* component forwards the results from *subBSCU1* by default. If *subBSCU1* reports that fault occurred through *Valid* signal, then *SelectSwitch* component forwards the results from *subBSCU2* subsystem.

8.3 Overall Development Approach

In order to make the safety contracts more useful, i.e., applicable in more different contexts and less susceptible to changes, we use the concept of safety kernels in the development process. Safety kernels are generally simple and independent mechanisms which behaviour can be easily ensured. Due to their simplicity and high independence, safety kernel behaviours can be specified more abstractly, i.e., with fewer context-specific assumptions. A reduced number of required assumptions increases reusability of safety information captured by the contracts. This allows us to provide better support for incremental certification through reuse of evidence and safety reasoning related to contracts, and ease change management within safety arguments. Besides safety kernels, other types of failure mitigation and recovery techniques can be implemented and packaged together with components. We refer to such techniques as component wrappers.

We build our development approaches that utilise the notions of safety kernels and component wrappers on the well-established practices recommended by safety standards. The proposed development approaches can be summarised by the following steps:

1. Perform a hazard analysis as required by most standards.

2. Perform causal analysis (e.g., FTA) to understand how the hazards can occur.

3. Create strong contracts for the fault handling behaviours that are offered in all contexts. Such behaviours that are specified more abstractly can be achieved with the use of safety kernels.

4. Create weak contracts for the fault handling behaviours that are context specific. Such behaviours are usually achieved by failure mitigation and recovery techniques (e.g., component wrappers) that are not developed with high independence from the context.

5. Create an architecture which includes:
   (a) Features to enforce the separation between the safety kernel and components. The safety kernel can only provide sufficient protection to allow it to provide fault tolerance if it can be argued that failures of the components do not interfere with its operation.
   (b) A design for the safety kernel that provides fault tolerance, principally fault detection and recovery, with respect to the mitigation of the more critical hazards.
   (c) A design for component wrappers that provides fault tolerance, principally fault detection and recovery, with respect to the mitigation of the less critical hazards. This largely deals with signal validation for data flowing in and out of the component. It is noted that some signals will be protected by both a wrapper and a safety kernel where used by multiple components.
6. Revise the fault tree to include the safety kernel and wrapper in the possible causes of hazards and judge whether the residual risks are acceptable. If the risks are not acceptable, judge whether more complex wrappers or more safety kernel functions would address the issues.

The development approach follows a typical set of stages except for the addition of contracts and the use of a safety kernel and wrappers. After deriving the safety contracts, the development approach continues to revise the contracts by checking if they are sufficiently complete and whether the described behaviours are sufficient to show that all identified hazards have been adequately addressed. Additional evidence backing the contracts is provided during the verification steps.

8.4 Definition of Safety Contracts

In this section we present part of the FTA performed on WBS with (section 8.4.2) and without (section 8.4.1) safety kernels. Later in section 8.4.3, we show how the results of the analysis can be used to derive safety contracts capturing corresponding safety behaviour of components addressed within the fault trees. In the second part of section 8.4.3 we discuss the problem of incompleteness of the safety contracts and propose how the contract completeness checking could be addressed.

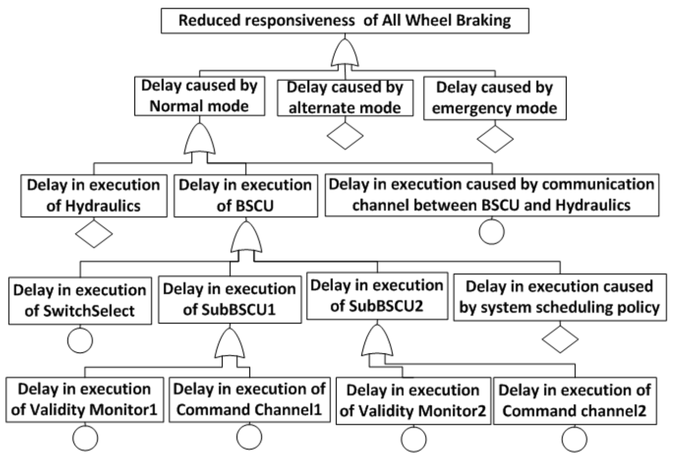

Reduced responsiveness of All Wheel Braking
Delay caused by Normal mode
Delay caused by alternate mode
Delay caused by emergency mode
Delay in execution of Hydraulics
Delay in execution of BSCU
Delay in execution caused by communication channel between BSCU and Hydraulics
Delay in execution of SwitchSelect
Delay in execution of SubBSCU1
Delay in execution of SubBSCU2
Delay in execution caused by system scheduling policy
Delay in execution of Validity Monitor1
Delay in execution of Command Channel1
Delay in execution of Validity Monitor2
Delay in execution of Command channel2

Figure 8.2: Reduced responsiveness of all wheel braking Fault Tree

8.4.1 Causal Analysis and Contracts for WBS

This section reuses the existing safety assessment of WBS presented in Appendix L of the ARP4761 document. Building upon the existing hazard analysis from Appendix L, we identified failure condition *reduced responsiveness of wheel braking* as hazardous, e.g., when it occurs during taxi phase it can lead to low-speed vehicle collision.

In order to prevent the delayed response from the brakes, we specify a timing safety requirement *SR1* that the WBS response time (i.e., time from the receipt of pedal brake signals to issuing the braking command) shall be no more than 10 ms. The fault tree in Fig. 8.2 addresses the reduced responsiveness failure condition. It shows that the delay in issuing the braking command can be caused by either of the three modes. The fault tree focuses on the normal mode and demonstrates that BSCU, Hydraulics or the communication channel between the two can all contribute to causing a delay in normal mode.

After identifying the hazards and specifying the requirements, the safety process continues to design the system to satisfy the specified requirements. Consequently, the safety contracts are captured to show compliance with the safety requirements. Strong safety contracts (denoted as a pair of strong assumptions and guarantees $\langle A, G \rangle$) allow us to specify behaviours that always must hold, i.e., strong assumptions (A) must be satisfied and strong guarantees (G) must be offered [6]. On the other hand, weak contracts (denoted as a pair of weak assumptions and guarantees $\langle B, H \rangle$) allow us to capture properties that change depending on the context in which the component is used. The weak guarantees (H) are offered only when all the strong contracts and the corresponding weak assumptions (B) are satisfied. The benefit of using the strong and weak contracts distinction is twofold: (1) it provides methodological distinction between properties that must hold and those that may hold in certain cases (e.g., weak contracts are used to describe multiple context-specific behaviours), and (2) when performing contract checking in a particular environment, violation of the strong assumptions is not tolerated, while violation of the weak assumptions is allowed (since some of the weak contracts might not be relevant for the particular context).

As the contracts need to be supported by evidence, we attach evidence information (E) with the contracts. We represent the contract/evidence pair as "C: $\langle A, G \rangle; E$", which can be read as follows: contract C, which under assumptions A offers guarantees G, is supported by evidence E. The motivation for connecting the evidence with the contracts is not to argue contract satisfaction (rationale description is needed for that), but to support change management. Besides identifying which parts of safety case are affected by change, safety contracts, when enriched by evidence information, can also be used to identify which evidence should be revisited. The evidence can be associated with a contract either directly, or indirectly through the associated contracts. Since the underlying contract formalism assumes hierarchical structure of components and contracts, all evidence needed to support a higher level contract are not associated with that contract directly, but can support the contract indirectly through the associated lower level contracts. The relation between a contract and its supporting contracts is established through the dependency assumptions.

Using component-based development notions, such as contracts, within safety-critical systems has some difficulties. The out-of-context idea of safety contracts causes difficulties that relate to both the nature of safety as a system property and context dependent behaviours such as timing [5]. When it comes to the nature of safety and contracts, it is difficult to capture all failure scenarios that the component can contribute to since what is safety relevant in one system might not be in another. For example, the difficulty with capturing timing properties within out-of-context contracts is not only that timing depends on many factors, but that the timing analysis is usually calculated with incompatible or simplified assumptions [14, 15, 16], which makes the timing information captured within contracts nearly impossible to reuse. While the inevitable solution

WBS_Weak_1:
$\langle$ **B1:** *Platform=x and Compiler=y* AND Hydraulics delay $\leq$ 4 ms AND BSCU delay $\leq$ 4 ms AND communication delay $\leq$ 0.1 ms AND emergency mode $\leq$ 1 ms;
H1: WBS delay $\leq$ 10 ms $\rangle$;
E1: WBS timing analysis under assumed conditions

Figure 8.3: WBS weak contract

WBS_BSCU_Weak_1:
$\langle$ **B2:** *Platform=x and Compiler=y* AND subBSCUx delay $<$ 3 ms and SelectSwitch delay $<$ 1 ms AND scheduler policy does not cause delay;
H2: BSCU delay $\leq$ 4 ms $\rangle$;
E2: BSCU timing analysis under assumed conditions

Figure 8.4: BSCU weak contract

in that case would be to re-run the timing analysis, the information captured within contracts can still be useful in highlighting impact of the change on the safety case.

Based on the causal analysis we specify the contract *WBS_Weak_1* (Fig. 8.3). *WBS_Weak_1* contains dependency assumptions capturing connection between WBS and its subcomponents, while the guaranteed property is the response time of WBS. In order to guarantee timing properties, such as those noted in [5], we need to include additional assumptions that are not provided in the causal analysis. In case of *WBS_Weak_1* contract we included additional assumptions on platform and compiler configuration, as such assumptions can be easily omitted from the causal analysis, and any change or inconsistency related to these properties may invalidate the corresponding contracts. We can note that the causal analysis is useful for capturing dependency assumptions within the safety contracts, but it is not sufficient as additional assumptions need to be captured as well. The Ariane 5 rocket is an example of how causality analysis does not cover some important assumptions. A piece of software that should perform certain computations right before liftoff was reused from the previous rocket version. Since restarting the software during liftoff might take time, the engineers decided to leave it running even after liftoff. The software then continued the unneeded computation during the flight time and caused an exception due to a floating-point error which rebooted the processor [17].

The contracts in Fig. 8.3 and 8.4 focus on the behaviour of WBS when there is no fault in the system. However the contracts don't describe behaviour of the

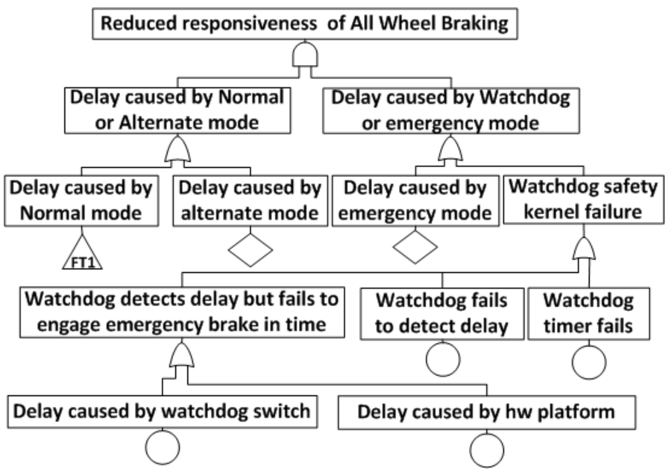

Reduced responsiveness of All Wheel Braking
Delay caused by Normal or Alternate mode
Delay caused by Watchdog or emergency mode
Delay caused by Normal mode
Delay caused by alternate mode
Delay caused by emergency mode
Watchdog safety kernel failure
FT1
Watchdog detects delay but fails to engage emergency brake in time
Watchdog fails to detect delay
Watchdog timer fails
Delay caused by watchdog switch
Delay caused by hw platform

Figure 8.5: Reduced responsiveness of all wheel braking Fault Tree (updated)

system in situation when anomalous behaviours occur, e.g. when BSCU delay is greater than 4 ms or the communication channel fails. As mentioned earlier, it is difficult to describe behaviour of a component in all the failure scenarios, e.g. in some cases it is reasonable to consider communication channel failure in others it may not be the case. While the described behaviour in contracts *WBS_Weak_1* and *WBS_BSCU_Weak_1* can be useful to know in certain situations, it is very difficult to reuse such information in case of platform change or moving the component from one system to another, as argued earlier. That is why this behaviour is specified within a weak contract, as it cannot be guaranteed in all systems. Further on we investigate how these weak contracts can be complemented with strong contracts capturing behaviour that prevents bad things from happening that is guaranteed wherever the component is used.

8.4.2 Causal Analysis and Contracts on WBS with Safety Kernels

In the current design, the reduced responsiveness of WBS can be caused by either of the modes. In order to reduce the criticality of timing requirements in the Normal and Alternate modes to an appropriate level, a design decision was made to use a simple and sufficiently independent safety kernel. This safety kernel acts as a last resort failure mechanism in case of failures that might pre-

WBS_Strong_1:
$\langle$ **A1:** sufficient independence of the safety kernel and emergency brake from normal and alternate mode
G1: if braking command not received from normal or alternate mode before watchdog timer expiry then kernel activates the emergency brake $\rangle$;
E3: Causal analysis; Contract completeness report
WBS_Weak_1:
$\langle$ **B1:** (*Platform=x and Compiler=y* AND Hydraulics delay $\leq$ 4 ms AND BSCU delay $\leq$ 4 ms AND communication delay $\leq$ 0.1 ms) OR (watchdog timer expiry $\leq$ 9 ms and emergency brake delay $\leq$1 ms);
H1: WBS delay $\leq$ 10 ms $\rangle$;
E1: New WBS timing analysis under assumed conditions

Figure 8.6: WBS contracts

WBS_SKC_Strong_1:
$\langle$ **A2:** -
G2: if the braking command signal not provided within 9 ms from the receipt of the pedal signals, then activate emergency brake within 1 ms $\rangle$;
E4: Formal verification report

Figure 8.7: Safety Kernel strong contract

vent Normal or Alternate mode from generating the braking command. The safety kernel in form of a watchdog timer is installed within Hydraulics component. Once WBS receives the pedal signals the watchdog timer is started. Unless either Normal or Alternate mode does not provide the braking command within the required time interval, the watchdog timer engages the emergency brake.

With introduction of the safety kernel in the WBS architecture, the initial FTA needs to be revisited to address both: changes to the criticality of Normal and Alternate modes; and extension of the current fault tree to include possible faults related to the kernel itself. The updated fault tree is shown in Fig. 8.5. The changes in the fault tree consequently influence the contracts to be revisited. More specifically, the *WBS_Weak_1* contract needs to be updated with the new information relating to the watchdog timer and the emergency brake. The updated *WBS_Weak_1* contract is shown in Fig. 8.6.

When using the safety kernels, we focus on capturing with the contracts how the component handles faults in the system. Due to simplicity of the kernel and its high independence from the rest of the system, we can specify strong safety contracts for the kernel that are easier to satisfy because of fewer assumptions. The strong contracts in Fig. 8.6 and 8.7 complement the weak contracts in Figures 8.3 and 8.4 by describing behaviour of the safety kernel when the normal or alternate mode fail. The assumption of sufficient independence in the contract *WBS_Strong_1* can be identified through the AND connection in the fault tree in Fig. 8.5 between *normal or alternate mode delays* and *kernel and emergency mode delays*. The corresponding guarantee describes the behaviour of the kernel in that situation. The *WBS_SKC_Strong_1* contract on the safety kernel addresses possible delay because of the kernel itself by guaranteeing its timing behaviour for all systems in which the kernel is used.

This example demonstrates that for the safety kernels we can specify the strong safety contracts with fewer assumptions (due to the simplicity and independence of the safety kernel). Fewer assumptions means that the corresponding contracts are easier to satisfy. Moreover, by reducing criticality of requirements addressed by the weak contracts, the stringency of evidence required to support the weak contracts is reduced. Consequently, overall less effort should be required for producing evidence to support such weak contracts.

8.4.3 Contract Derivation and Completeness Checking Methods

To talk about completeness of contracts we need to identify with respect to what should that completeness be checked. The safety contracts focus on failure behaviours of the system that can be obtained by failure analysis (e.g., FTA) as these are most often the causes of hazards. In this work we use FTA, a well-established method recommended by safety standards, for contract derivation and completeness check. Deriving contracts from fault trees is performed as follows:

1. Identify fault tree nodes directly related to the component for which the contract is being derived such that the nodes do not belong to each others sub-branches.
2. For each identified node:
   (a) Create a safety contract that guarantees to prevent or minimise the faulty state described by the node.
   (b) Identify candidate nodes for stating dependency assumptions such that the assumption node belongs to the same branch as the guaran-

tee node, and that it refers to behaviour either of first level subcomponent of the current component, other components in the environment that the current component is connected to or other system properties.

3. The logical connection of the assumptions within the contract is switched comparing to the connection in the fault tree (e.g., AND connections in the fault trees become OR in the contracts), similarly as the guarantees can be regarded as negations of the corresponding nodes (e.g., a node "delay in execution" in a fault tree becomes a guarantee "does not cause delay in execution").

The assumptions on the first-level subcomponents are included to capture dependencies between the two layers identified by FTA, and in that way facilitate independent development and change management. For example, *BSCU* is independently developed by a contractor. Based on the specified dependency assumptions we can identify if the provided (or replaced) component offers required behaviour to achieve the WBS behaviour. This can be done by checking if the WBS dependency assumptions are satisfied by BSCU contracts.

Once the change occurs in the system or the component is moved to another system, the completeness of the contracts needs to be checked with respect to the fault trees. In our case, the contract *WBS_Weak_1* had to be changed after introducing the safety kernel as the contract was not complete with respect to the new fault tree in Fig. 8.5. Consequently, the evidence required to support this contract had to be updated.

Completeness with respect to a specific failure analysis does not imply contract completeness in general, but only with respect to the analysis. Confidence in the completeness check stems from the confidence in the failure analysis against which the check is performed. In our work we use FTA for completeness check under assumptions that producing fault trees is well-established and that the resulting fault trees are reasonably complete. It must be emphasised that the approach does not rely on the fault trees actually being complete, as the aim is to de-risk change rather than have a change process where only contracts have to be checked following a change. The derived contracts usually require additional assumptions that can be derived from different analyses and used to enrich the contracts, hence increase their overall completeness. The contracts completeness check with respect to a specific analysis is performed to ensure that there are no inconsistencies between the dependencies captured within the contracts and those identified by the analysis. The results of such check can indicate that the contracts are incomplete with respect to the analysis (in case of changes to the system, and to the analysis), or the analysis can be incomplete with respect to the contracts (if we have enriched the contracts using other types of analyses). The contract completeness check with respect to the fault trees is performed as follows:

1. Identify nodes in the fault tree that correspond to the contract guarantees.

2. Identify nodes in the fault tree corresponding to the assumptions.

3. For the identified assumptions within the fault tree, check whether they belong to the branch corresponding to the identified node related to the guarantee.

4. Identify the following inconsistencies:

   (a) Nodes that are included in the assumptions but do not belong to the same branch as the guarantee node.

   (b) Nodes within the same branch as the guaranteed node that are not covered by the assumptions (not all nodes of the branch should be captured by assumptions but all should be covered, i.e., if the node itself is not included, then its sub-nodes or leaves of its branch should be included for the node to be covered).

5. If assumptions cover all nodes within the guarantees node branch then the contract is complete with respect to the fault tree, but if there are additional nodes that are assumed but do not belong to the same branch, the inconsistency should be reported as either fault tree is not complete, or the contract should be revised.

8.5 Safety Argument

In this section we present an overview of the graphical notation (section 8.5.1) used to construct our arguments. The WBS safety argument is presented in section 8.5.2.

8.5.1 Overview of Goal Structuring Notation

The Goal Structuring Notation (GSN) [18] – a graphical argumentation notation – explicitly represents the individual elements of any safety argument (requirements, claims, evidence and context) and (perhaps more significantly) the

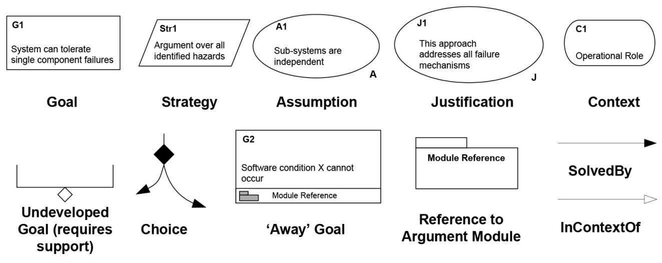

G1
System can tolerate single component failures
Goal
Str1
Argument over all identified hazards
Strategy
A1
Sub-systems are independent
A
Assumption
J1
This approach addresses all failure mechanisms
J
Justification
C1
Operational Role
Context
Undeveloped Goal (requires support)
Choice
G2
Software condition X cannot occur
Module Reference
'Away' Goal
Module Reference
Reference to Argument Module
SolvedBy
InContextOf

Figure 8.8: Overview of the Goal Structuring Notation (GSN)

relationships that exist between these elements (i.e. how individual requirements are supported by specific claims, how claims are supported by evidence and the assumed context that is defined for the argument). The principal symbols of the notation are shown in Fig. 8.8 (with example instances of each concept).

The principal purpose of a goal structure is to show how goals (claims about the system) are successively broken down into ("solved by") sub-goals until a point is reached where claims can be supported by direct reference to available evidence. As part of this decomposition, using the GSN it is also possible to make clear the argument strategies adopted (e.g. adopting a quantitative or qualitative approach), the rationale for the approach (assumptions, justifications) and the context in which goals are stated (e.g. the system scope or the assumed operational role). For further details on GSN see [18]. GSN has been widely adopted by safety-critical industries for the presentation of safety arguments within safety cases. While GSN is mainly used to record monolithic safety arguments, an extension facilitates the creation of modular arguments. As a part of the modularised form of GSN, an away goal statement can be used to support the local claim by referring to a claim developed in another module. In this paper the modularised form of GSN, as first introduced in [19, 20], is used.

8.5.2 Wheel Braking System Safety Argument

Fig. 8.9 shows the safety argument fragment for WBS represented using GSN. The argument focuses on the timing requirement *SR1*: "WBS response time shall be no more than 10 ms", specified in Section 8.4 and represented by the goal *WBSSafetyExeTime* within the argument. We base the argument that

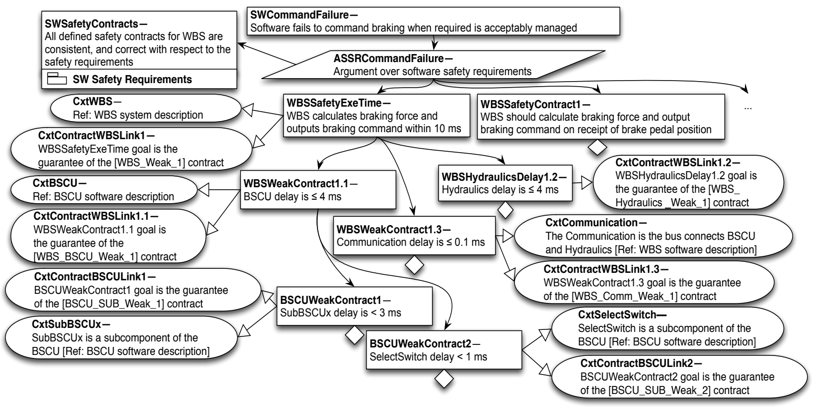

SWSafetyContracts—
All defined safety contracts for WBS are consistent, and correct with respect to the safety requirements
SW Safety Requirements
SWCommandFailure—
Software fails to command braking when required is acceptably managed
ASSRCommandFailure—
Argument over software safety requirements
CxtWBS—
Ref: WBS system description
CxtContractWBSLink1—
WBSSafetyExeTime goal is the guarantee of the [WBS_Weak_1] contract
WBSSafetyExeTime—
WBS calculates braking force and outputs braking command within 10 ms
WBSSafetyContract1—
WBS should calculate braking force and output braking command on receipt of brake pedal position
...
CxtBSCU—
Ref: BSCU software description
CxtContractWBSLink1.1—
WBSWeakContract1.1 goal is the guarantee of the [WBS_BSCU_Weak_1] contract
WBSWeakContract1.1—
BSCU delay is ≤ 4 ms
WBSHydraulicsDelay1.2—
Hydraulics delay is ≤ 4 ms
CxtContractWBSLink1.2—
WBSHydraulicsDelay1.2 goal is the guarantee of the [WBS_ Hydraulics _Weak_1] contract
WBSWeakContract1.3—
Communication delay is ≤ 0.1 ms
CxtCommunication—
The Communication is the bus connects BSCU and Hydraulics [Ref: WBS software description]
CxtContractWBSLink1.3—
WBSWeakContract1.3 goal is the guarantee of the [WBS_Comm_Weak_1] contract
CxtContractBSCULink1—
BSCUWeakContract1 goal is the guarantee of the [BSCU_SUB_Weak_1] contract
CxtSubBSCUx—
SubBSCUx is a subcomponent of the BSCU [Ref: BSCU software description]
BSCUWeakContract1—
SubBSCUx delay is < 3 ms
BSCUWeakContract2—
SelectSwitch delay < 1 ms
CxtSelectSwitch—
SelectSwitch is a subcomponent of the BSCU [Ref: BSCU software description]
CxtContractBSCULink2—
BSCUWeakContract2 goal is the guarantee of the [BSCU_SUB_Weak_2] contract

Figure 8.9: WBS safety argument before introducing the safety kernel

SR1 is satisfied on the *WBSSWSafetyReq* justification that the software safety requirements are addressed by the safety contracts. Moreover we provide an away goal *SWSafetyContracts* presenting the required evidence to support safety contract consistency, their correctness with respect to the associated safety requirements and completeness with respect to the failure analysis. In the presented argument we focus on the product rather than the process by which we ensure that these contract properties are achieved.

Based on the *WBSSWSafetyReq* justification we address the *WBSSafetyExeTime* goal by the *WBS_weak_1* contract that supports the *SR1* requirement. In order to clarify the *WBSSafetyExeTime* goal, we create a context statement to identify the *WBS_weak_1* contract that addresses the goal, and to provide a reference to WBS system description. To further develop the *WBSSafetyExeTime* goal, we use the dependency assumptions of the associated contract *WBS_weak_1* to identify the supporting sub-goals: *WBSWeakContract1.1*, *WBSHydraulicsDelay1.2* and *WBSWeakContract1.3*. The context statements for these sub-goals are provided in the same way as for the *WBSSafetyExeTime* goal. Further development of the sub-goals follows the same procedure as for the *WBSSafetyExeTime* goal, i.e. by identifying dependency assumptions of the associated contract to the particular goal, we derive sub-goals until we reach the lowest level component, i.e. where we have directly relevant evidence that supports the goal.

As WBS architecture changed with addition of the safety kernel, the cor-

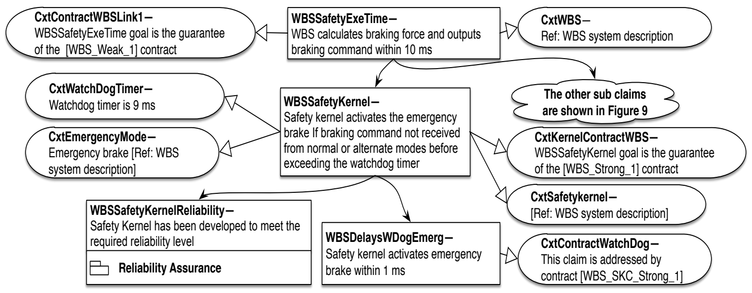

CxtContractWBSLink1 –
WBSSafetyExeTime goal is the guarantee of the [WBS_Weak_1] contract
WBSSafetyExeTime –
WBS calculates braking force and outputs braking command within 10 ms
CxtWBS –
Ref: WBS system description
CxtWatchDogTimer –
Watchdog timer is 9 ms
WBSSafetyKernel –
Safety kernel activates the emergency brake If braking command not received from normal or alternate modes before exceeding the watchdog timer
The other sub claims are shown in Figure 9
CxtEmergencyMode –
Emergency brake [Ref: WBS system description]
CxtKernelContractWBS –
WBSSafetyKernel goal is the guarantee of the [WBS_Strong_1] contract
CxtSafetykernel –
[Ref: WBS system description]
WBSSafetyKernelReliability –
Safety Kernel has been developed to meet the required reliability level
Reliability Assurance
WBSDelaysWDogEmerg –
Safety kernel activates emergency brake within 1 ms
CxtContractWatchDog –
This claim is addressed by contract [WBS_SKC_Strong_1]

Figure 8.10: The updated WBSSafetyExeTime goal after introducing the safety kernel

responding safety argument needs to be updated as well. Based on the derived safety contracts for the safety kernel provided in Figures 8.6 and 8.7, we extend the safety argument from Fig. 8.9 with an additional supporting goal *WBSSafetyKernel* to the *WBSSafetyExeTime* claim, as shown in Fig. 8.10. The goal *WBSSafetyKernel* is clarified with context statements by referring to the corresponding contract *WBS_Strong_1* (Fig. 8.6), and providing definitions of the timer interval of 9 ms, and notions of emergency brake and safety kernel definition. The *WBSSafetyKernel* goal is further supported by an away goal *WBSSafetyKernelReliability* claiming that the kernel has been developed to meet the required reliability level, and a sub-goal *WBSDelaysWDogEmerg* based on the *WBS_SKC_Strong_1* contract.

paperC

8.6 Summary and Conclusions

Means to capture failure behaviour within safety contracts have received little attention in contract-based approaches for safety-critical systems. Moreover, handling of inevitable contract incompleteness, implied by a great number of assumptions that need to be captured, is not sufficient for showing that the system is acceptably safe. We have presented a method for deriving safety contracts from fault tree analysis and demonstrated it on an example. Once the initial contracts have been derived, we introduced a safety kernel to the system architecture to reduce the criticality of the rest of the system. To handle the change in the system, we have proposed that completeness of the contracts derived from failure analysis is re-evaluated with respect to that analysis after the change has been introduced and the analysis updated. The proposed completeness check method identifies inconsistencies between the contracts and the failure analysis and acts as guidance for change management. We have used the notion of safety kernels to show how strong safety contracts can be derived with fewer assumptions due to kernel's simplicity and high independence from the rest of the system. Deriving contracts from failure analysis results in at least as complete contracts as the analysis itself. While particular analysis itself can be incomplete, different analyses can be used to enrich the contracts and increase their completeness.

Future work will focus on developing safety contract-based change management techniques, which should cover both the safety argument and associated evidence. Furthermore, we plan to investigate techniques for identifying additional assumptions needed to enrich the contracts derived from failure analysis.

Acknowledgement

We acknowledge the Swedish Foundation for Strategic Research (SSF) SYNOPSIS Project.

Bibliography

[1] Albert Benveniste, Benot Caillaud, Alberto Ferrari, Leonardo Mangeruca, Roberto Passerone, and Christos Sofronis. Multiple viewpoint contract-based specification and design. In *Formal Methods for Components and Objects*, volume 5382 of *Lecture Notes in Computer Science*, pages 200–225, 2007.

[2] Werner Damm, Hardi Hungar, Bernhard Josko, Thomas Peikenkamp, and Ingo Stierand. Using contract-based component specifications for virtual integration testing and architecture design. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2011.

[3] Alessandro Cimatti and Stefano Tonetta. A property-based proof system for contract-based design. In *Proceedings of the 38th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 21–28. IEEE Computer Society, 2012.

[4] J. L. Fenn, R. Hawkins, P. J. Williams, T. Kelly, M. G. Banner, and Y. Oakshott. The Who, Where, How, Why and When of Modular and Incremental Certification. In *2nd Institution of Engineering and Technology International Conference on System Safety*, pages 135–140. IET, 2007.

[5] Patrick Graydon and Iain Bate. On the nature and content of safety contracts. In *Proceedings of the 15th IEEE International Symposium on High Assurance Systems Engineering*, pages 245–246, January 2014.

[6] Irfan Sljivo, Barbara Gallina, Jan Carlson, and Hans Hansson. Strong and weak contract formalism for third-party component reuse. In *Proceedings of the 3rd International Workshop on Software Certification*, pages 359–364, 2013.

[7] Modular software safety case (MSSC) — process description. https://www.amsderisc.com/related-programmes/, Nov 2012.

[8] J. Rushby. *Kernels for Safety?*, chapter 13, pages 210–220. Blackwell Scientific Publications, 1989.

[9] K. Wika and J. Knight. On The Enforcement Of Software Safety Policies. In *10th Annual Conference on Computer Assurance*. IEEE, June 1995.

[10] Bastian Zimmer, Susanne Bürklen, Michael Knoop, Jens Höfflinger, and Mario Trapp. Vertical safety interfaces–improving the efficiency of modular certification. In *Proceedings of the 30th International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*, pages 29–42. 2011.

[11] T. Kelly and J. McDermid. A Systematic Approach to Safety Case Maintenance. *Reliability Engineering and System Safety*, 71(3):271 – 284, 2001.

[12] O. Jaradat, P. Graydon, and I. Bate. An Approach to Maintaining Safety Case Evidence After A System Change. In *10th European Dependable Computing Conference (EDCC)*, August 2014.

[13] SAE ARP4761 Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, December 1996.

[14] Patrick Graydon and Iain Bate. Realistic safety cases for the timing of systems. *The Computer Journal*, 57(5):759–774, May 2014.

[15] T. Kelly, I. Bate, J. McDermid, and A. Burns. Building a Preliminary Safety Case: An Example from Aerospace. In *Australian Workshop on Industrial Experience with Safety Critical Systems and Software*, October 1997.

[16] I. Bate and A. Burns. An Integrated Approach to Scheduling in Safety-Critical Embedded Control Systems. *Real-Time Systems Journal*, 25(1):5–37, July 2003.

[17] Jean-Marc Jézéquel and Bertrand Meyer. Design by contract: The lessons of ariane. *IEEE Computer*, 30(1):129–130, January 1997.

[18] T. Kelly. *Arguing Safety — A Systematic Approach to Managing Safety Cases*. PhD thesis, University of York, York, UK, 1998.

[19] I. Bate and T. Kelly. Architectural Considerations in the Certification of Modular Systems. In *21st International Conference on Computer Safety, Reliability, and Security*, volume 2434 of *Lecture Notes in Computer Science*, pages 321–333. Springer, 2002.

[20] I. Bate and T. Kelly. Architectural Considerations in the Certification of Modular Systems. *Reliability Engineering and System Safety*, 81(3):303–324, 2003.

Chapter 9

Paper D: Using Sensitivity Analysis to Facilitate The Maintenance of Safety Cases

Omar Jaradat, Iain Bate, Sasikumar Punnekkat.
In Proceedings of the 20th International Conference on Reliable Software Technologies (Ada-Europe 2015), Madrid, Spain, June 2015.

Abstract

A safety case contains safety arguments together with supporting evidence that together should demonstrate that a system is acceptably safe. System changes pose a challenge to the soundness and cogency of the safety case argument. Maintaining safety arguments is a painstaking process because it requires performing a change impact analysis through interdependent elements. Changes are often performed years after the deployment of a system making it harder for safety case developers to know which parts of the argument are affected. Contracts have been proposed as a means for helping to manage changes. There has been significant work that discusses how to represent and to use them but there has been little on how to derive them. In this paper, we propose a sensitivity analysis approach to derive contracts from Fault Tree Analyses and use them to trace changes in the safety argument, thus facilitating easier maintenance of the safety argument.

9.1 Introduction

Building a safety case is an increasingly common practice in many safety critical domains [1]. A safety case comprises both safety evidence and a safety argument that explains that evidence. The safety evidence is collected throughout the development and operational phases, for example from analysis, test, inspection, and in-service monitoring activities. The safety argument shows how this evidence demonstrates that the system satisfies the applicable operational definition of acceptably safe to operate in its intended operating context.

A safety case should always justify the safety status of the associated system, therefore it is described as a living document that should be maintained as needed whenever some aspect of the system, its operation, its operating context, or its operational history changes. However, safety goals, evidence, argument, and assumptions about operating context are interdependent and thus, seemingly-minor changes may have a major impact on the contents and structure of the safety argument. Any improper maintenance in a safety argument has a potential for a tremendous negative impact on the conveyed system safety status by the safety case. Hence, a step to assess the impact of this change on the safety argument is crucial and highly needed prior to updating a safety argument after a system change.

Changes to the system during or after development might invalidate safety evidence or argument. Evidence might no longer support the developers' claims because it reflects old development artefacts or old assumptions about operation or the operating environment. In the updated system, existing safety claims might not make any sense, no longer reflect operational intent, or be contradicted by new data. Analysing the impact of a change in a safety argument is not trivial: doing so requires awareness of the dependencies among the argument's contents and how changes to one part might invalidate others. In other words, if a change was applied to any element of a set of interdependent elements, then the associated effects on the rest of the elements might go unnoticed. Without this vital awareness, a developer performing impact analysis might not notice that a change has compromised system safety. Implicit dependencies thus pose a major challenge. Moreover, evidence is valid only in the operational and environmental contexts in which it was obtained or to which it applies. Operational or environmental changes might therefore affect the relevance of evidence and, indirectly, the validity and strength of the safety argument.

Predicting system changes before building a safety argument can be useful because it allows the safety argument to be structured to contain the impact of these changes. Hence, anticipated changes may have predictable and traceable consequences that will eventually reduce the maintenance efforts. Nevertheless, planning the maintenance of a safety case still faces two key issues: (1) system changes and their details cannot be fully predicted and made available up front, especially, the software aspects of the safety case as software is highly changeable and harder to manage as they are hard to contain, and (2) those changes can be implemented years after the development of a safety case. Part of what we aim for in this work is to provide system developers a list of system parts that may be more problematic to change than other parts and ask them to choose the parts that are most likely to change. Of course our list can be augmented by additional changeable parts that may be provided by the system developers.

Sensitivity analysis helps the experts to define the uncertainties involved with a particular system change so that those experts can judge on the potential change based on how reliable they feel the consequences are. The analysis can deal with what aim for since it allows us to define the problematic changes. More specifically, we exploit the Fault Tree Analyses (FTAs) which are supposed to have been done by developers through the safety analysis phase and apply the sensitivity analysis to those FTAs in order to identify the sensitive parts in them. We define a sensitive part as one or multiple events whose minimum changes have the maximal effect on the FTA, where effect means exceeding reliability targets due to a change.

In spite of the assumption we make that the safety argument's logic is based on the causal pathways described in the FTAs, tracking the changes from the FTAs of a system down to its safety argument still requires a traceability mechanism between the two. To this end, we use the concept of contract to highlight the sensitive parts in FTAs, and to establish a traceability between those parts and the corresponding safety argument. In our work, we assume that safety arguments are recorded in the Goal Structuring Notation (GSN) [2]. However, the approach we propose might (with suitable adaptations) be compatible for use with other graphical assurance argument notations.

Combining the sensitivity analysis together with the concept of contracts to identify the sensitive parts of a system and highlight these parts may help the experts to make an educated decision as to whether or not apply changes. This decision is in light of beforehand knowledge of the impact of these changes on the system and its safety case. Our hypothesis in this work is that it is possible to use the sensitivity analysis together with safety contracts to (1) bring to developers' attention the most sensitive parts of a system for a particular change, and (2) manage the change by guiding the developers to the parts in the safety argument that might be affected after applying a change. However, using contracts as a way of managing change is not a new notion since it has been discussed in some works, such as [3][4], but deriving the contracts and their contents have received little or even no support yet. The main contribution of this paper is to propose a safety case maintenance technique. However, we focus on the first phase of the technique and explain how to apply the sensitivity analysis to FTAs and derive the contracts and their contents. We also explain how to associate the derived arguments with safety argument goals. The paper illustrates the technique and its key concepts using the a hypothetical aircraft Wheel Braking System (WBS).

The paper is structured as follows: in Section 9.2 we present background information. In Section 9.3 we propose a technique for maintaining safety cases using sensitivity analysis. In Section 9.4 we use the WBS example to illustrate the technique. In Section 9.5 we present the related work. Finally, we conclude and derive future works in Section 9.6.

9.2 Background and Motivation

This section gives background information about (1) the GSN, (2) the concept of contract, (3) some of the current challenges that are facing safety case maintenance including a brief review of the state-of-the-art, and (4) the sensitivity analysis including some possible applications.

9.2.1 The Goal Structuring Notation (GSN)

A safety argument organizes and communicates a safety case, showing how the items of safety evidence are related and collectively demonstrate that a system is acceptably safe to operate in a particular context. GSN [2] provides a graphical means of communicating (1) safety argument elements, claims (goals), argument logic (strategies), assumptions, context, evidence (solutions), and (2) the relationships between these elements. The principal symbols of the notation are shown in Figure 9.1 (with example instances of each concept).

A goal structure shows how goals are successively broken down into ('solved by') sub-goals until eventually supported by direct reference to evidence. Using the GSN, it is also possible to clarify the argument strategies adopted (i.e., how the premises imply the conclusion), the rationale for the approach (assumptions, justifications) and the context in which goals are stated.

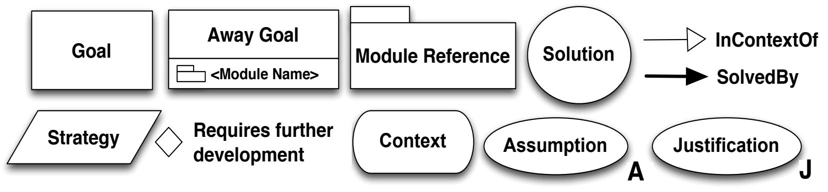


Figure 9.1: Notation Keys of the Goal Structuring Notation (GSN)

9.2.2 The Concept of Safety Contracts

The concept of contract is not uncommon in software development and it was first introduced in 1988 by Meyer [5] to constrain the interactions that occur between objects. Contract-based design [6] is defined as an approach where the design process is seen as a successive assembly of components where a component behaviour is represented in terms of assumptions about its environment and guarantees about its behavior. Hence, contracts are intended to describe functional and behavioral properties for each design component in form of assumptions and guarantees. In this paper, a contract that describes properties that are only safety-related is referred to as a safety contract.

9.2.3 Safety Case Maintenance and Current Practices

A safety case is a living document that should be maintained as the system, its operation, or its operating context changes. In this paper, we refer to the process of updating the safety case after implementing a change as safety case maintenance. Developers are experiencing difficulties with safety case maintenance, including difficulty identifying the direct and indirect impact of change. Two main causes of this difficulty are a lack of traceability between a system and its safety case and a lack of documentation of dependencies among the safety case's contents. Systems tend to become more complex, this increasing complexity can exacerbate safety case maintenance difficulties. The GSN is meant to reduce these difficulties by providing a clear, explicit conceptual model of the safety case's elements and interdependencies [7].

Our discussion of documenting interdependencies within a safety case refers to two different forms of traceability. Firstly, we refer to the ability to relate safety argument fragments to system design components as component traceability (through a safety argument). Secondly, we refer to evidence across system's artefacts as evidence traceability.

Current standards and analysis techniques assume a top-down development approach to system design. When systems that are built from components, mismatch with design structure makes monolithic safety arguments and evidence difficult to maintain. Safety is a system level property; assuring safety requires safety evidence to be consistent and traceable to system safety goals [7]. One might suppose that a safety argument structure aligned with the system design structure would make traceability clearer. It might, but safety argument structures are influenced by four factors: (1) modularity of evidence, (2) modularity of the system, (3) process demarcation (e.g., the scope of ISO 26262 items [1]), and organisational structure (e.g., who is working on what). These factors often make argument structures aligned with the system design structure impractical. However, the need to track changes across the whole safety argument is still significant for maintaining the argument regardless of its structure.

9.2.4 Sensitivity Analysis

Sensitivity analysis helps to establish reasonably acceptable confidence in the model by studying the uncertainties that are often associated with variables in models. There are different purposes for using sensitivity analysis, such as, providing insight into the robustness of model results when making decisions [8]. The analysis can be also used to enhance communication from modelers to decision makers, for example, by making recommendations more credible, understandable, compelling or persuasive [9]. The analysis can be performed by different methods, such as, mathematical, graphical, statistical, etc.

In this paper, we use sensitivity analysis to identify the sensitive parts of a system that might require unnecessary painstaking maintenance. More specifically, we apply the sensitivity analysis on FTAs to measure the sensitivity of outcome A (e.g., a safety requirement being true) to a change in a parameter B (e.g., the failure rate in a component). The sensitivity is defined as $\Delta B/B$, where ΔB is the smallest change in B that changes A (e.g., the smallest increase in failure rate that makes safety requirement A false). Hence, a sensitive part is defined as one or multiple FTA events whose minimum changes have the maximal effect on the FTA, where effect means exceeding failure probabilities (reliability targets) to inadmissible levels due to the change. The failure probability values that are attached to the FTA events are considered input parameters to the sensitivity analysis. A sensitive event is the event whose failure probability value can significantly influence the validity of the FTA once it increases. A sensitive part of a FTA is assigned to a system design component that is referred to as sensitive component in this paper. Hence, changes to a sensitive component cause a great impact to system design.

9.3 Using Sensitivity Analysis To Facilitate The Maintenance of A Safety Case

In this section, we build on the background information provided in Section 9.2 to propose a technique that aims to facilitate the maintenance of a safety case. The technique comprises 7 steps that are distributed between the Sensitivity ANalysis for Enabling Safety Argument Maintenance (SANESAM) phase and the safety argument maintenance phases as shown in Figure 9.2. The steps of the SANESAM phase are represented along the upper path, whilst the lower path represents the steps of the safety argument maintenance phase. The SANESAM phase, however, is what is being discussed in this paper.

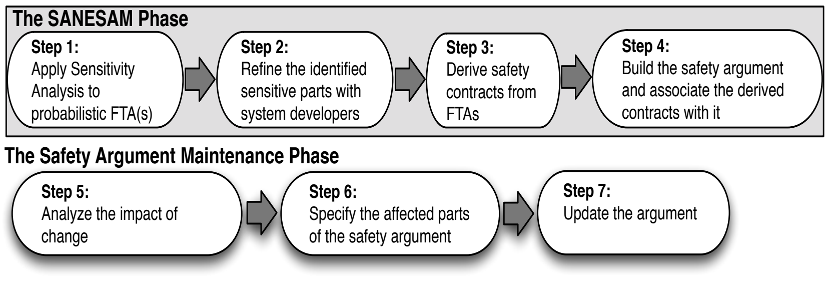

The SANESAM Phase
Step 1: Apply Sensitivity Analysis to probabilistic FTA(s)
Step 2: Refine the identified sensitive parts with system developers
Step 3: Derive safety contracts from FTAs
Step 4: Build the safety argument and associate the derived contracts with it
The Safety Argument Maintenance Phase
Step 5: Analyze the impact of change
Step 6: Specify the affected parts of the safety argument
Step 7: Update the argument

Figure 9.2: Process diagram of the proposed technique

A complete approach to managing safety case change would include both (a) mechanisms to structure the argument so as to contain the impact of predicted changes and (b) means of assessing the impact of change on all parts of the argument [10]. As discussed in Section 9.1, system changes and their details cannot be fully predicted and made available up front. Predicting potential changes to the software aspects of a safety case is even harder than other parts because software is highly changeable and harder to manage. Consequently, considering a complete list of anticipated changes is difficult. What can be easier though is to determine the flexibility (or compliance) of each component to changes. This means that regardless of the type of changes the latter will be seen as factors to increase or decrease a certain parameter value. Thus system developers can focus more on predicting those changes that might make the parameter value inadmissible.

The rationale of our technique is to determine, for each component, the allowed range for a certain parameter within which a component may change before it compromises a certain system property (e.g., safety, reliability, etc.). To this end, we use the sensitivity analysis as a method to determine the range of failure probability parameter for each component. Hence, the technique assumes the existence of a probabilistic FTA where each event in the tree is specified by an actual (i.e., current) failure probability $FP_{Actual|event(x)}$. In addition, the technique assumes the existence of the required failure probability for the top event $FP_{Required(Topevent)}$, where the FTA is considered unreliable if: $FP_{Actual(Topevent)} > FP_{Required(Topevent)}$. The steps of the SANESAM phase are as follows:

- ***Step 1. Apply the sensitivity analysis to a probabilistic FTA***: In this step the sensitivity analysis is applied to a FTA to identify the sensitive events whose minimum changes have the maximal effect on the $FP_{Topevent}$. Identifying those sensitive events requires the following steps to be performed:

  1. Find minimal cut set MC in the FTA. The minimal cut set definition is: *"A cut set in a fault tree is a set of basic events whose (simultaneous) occurrence ensures that the top event occurs. A cut set is said to be minimal if the set cannot be reduced without losing its status as a cut set"[11].*
  2. Calculate the maximum possible increment in the failure probability parameter of event x before the top event $FP_{Actual(Topevent)}$ is no longer met, where $x \in MC$ and
  $$(FP_{Increased|event(x)} - FP_{Actual|event(x)}) \nRightarrow FP_{Actual(Topevent)} > FP_{Required(Topevent)}.$$
  3. Rank the sensitive events from the most sensitive to the less sensitive. The most sensitive event is the event for which the following equation is the minimum:
  $$(FP_{Increased|event(x)} - FP_{Actual|event(x)})/FP_{Actual|event(x)}$$

- ***Step 2. Refine the identified sensitive parts with system developers***: In this step, the generated list from Step 1 should be discussed with system developers (e.g., safety engineers) and ask them to choose the sensitive events that are most likely to change. The list can be extended to add any additional events by the developers. Moreover, it is envisaged that some events may be removed from the list or the rank of some of them change.

- ***Step 3. Derive safety contracts from FTAs***: In this step, the refined list from Step 2 is used as a guide to derive the safety contracts, where each event in the list should have at least one contract. The main objective of the contracts is to 1) highlight the sensitive events to make them visible up front for developers attention, and 2) to record the dependencies between the sensitive events and the other events in the FTA. Hence, if any contracted event has received a change that necessitates increasing its failure probability where the increment is still within the defined threshold in the contract, then it can be said that the contract(s) in question still holds (intact) and the change is containable with no further maintenance. The contract(s), however, should be updated to the latest failure probability value. On the contrary, if the change causes a bigger increment in the failure probability value than the contract can hold, then the contract is said to be broken and the guaranteed event will no longer meet its reliability target. We create a template to document the derived safety contracts as shown in Figure 9.3a, where G and A stand for Guarantee and Assumption, respectively. Furthermore, each safety contract should contain a version number (it is shown as **V** in Figure 9.3a). The version number of the contract should match the *artefact version number* (as described in the next step), otherwise it will be considered out of date. We also introduce a new notation to the FTA to annotate the contracted events where every created contract should have a unique identifier, see Figure 9.3b.

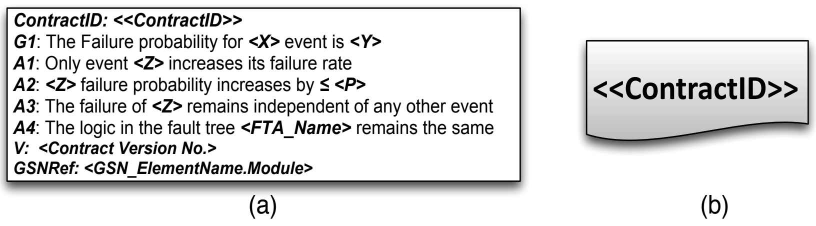
- ***Step 4. Build the safety argument and associate the derived contracts with it***: In this step, a safety argument should be built and the derived safety contracts should be associated with the argument elements.

  In order to associate the derived safety contracts with GSN arguments, we reuse our previous work [10]. The essence of that work is storing additional information in the safety argument to facilitate identifying the evidence impacted by change. This is done by annotating each reference to a development artefact (e.g. an architecture specification) in a goal or context element with an *artefact version number*. Also by annotating each solution element with:

1. An *evidence version* number
2. An *input manifest* identifying the inputs (including version) from which the evidence was produced
3. The *lifecycle phase* during which the evidence obtained (e.g. Software Architecture Design)
4. A *safety standard reference* to the clause in the applicable standard (if any) requiring the evidence (and setting out safety integrity level requirements)

However, the approach description, just as it is, does not support associating our derived safety contracts in ***Step 3*** with the safety argument without proper adjustments. Hence, a set of rules are introduced to guide the reuse of the approach in the work of this paper, as follows:

1. GSN element names should be unique.
2. At least one GSN goal should be created for each guarantee (i.e., for each safety contract). Moreover, the contract should be annotated in the goal which is made for it. The annotation should be done by using the contract ID and the notation in Figure 9.3b.
3. Assumptions in each safety contract should be restricted to one event only. If the guarantee requires assumptions about another event, a new contract should be created to cover these assumptions.



ContractID: <<ContractID>>
G1: The Failure probability for ***<X>*** event is ***<Y>***
A1: Only event ***<Z>*** increases its failure rate
A2: ***<Z>*** failure probability increases by ≤ ***<P>***
A3: The failure of ***<Z>*** remains independent of any other event
A4: The logic in the fault tree ***<FTA_Name>*** remains the same
V: <Contract Version No.>
GSNRef: <GSN_ElementName.Module>

Figure 9.3: (a) Safety Contract Template (b) Safety Contract Notation for FTA

4. An event in the assumptions list of a safety contract may be also used as a goal in the argument. In this case, the goal name should be similar to the event name.
5. Each safety contract should contain the GSN reference within it. The reference is the unique name of the GSN element followed by a dot and the name of the GSN module (if modular GSN is used). It is worth noting that while documenting the safety contracts, the GSN references might not be available as the safety argument itself might not be built yet. Hence, whenever GSN references are made available, system developers are required to revisit each contract and add the corresponding GSN reference to it. GSN reference parameter is shown as **GSNRef** in Figure 9.3a.

It is worth saying that the technique shall not affect the way GSN is being produced but it brings additional information for developers' attention.

9.4 An Illustrative Example: The Wheel Braking System (WBS)

In this section, we illustrate the proposed technique and its key concepts using the hypothetical aircraft braking system described in Appendix L of Aerospace Recommended Practice ARP-4761 [12]. Figure 9.4 shows a high-level architecture view of the WBS

9.4.1 Wheel Braking System (WBS): System Description

The WBS is installed on the two main landing gears. The main function of the system is to provide wheel braking as commanded by the pilot when the aircraft is on the ground. The system is composed of three main parts: Computer-based part which is called the Brake System Control Unit (*BSCU*), *Hydraulic* part, and *Mechanical* part.

The *BSCU* is internally redundant and consists of two channels, *BSCU System 1 and 2* (*BSCU* is the box in the gray background in Figure 9.4). Each channel consists of two components: *Monitor* and *Command*. *BSCU System 1 and 2* receive the same pedal position inputs, and both calculate the command value. The two command values are individually monitored by the *Monitor 1 and 2*. Subsequently, values are compared and if they do not agree, a failure is reported. The results of both *Monitors* and the compared values are provided

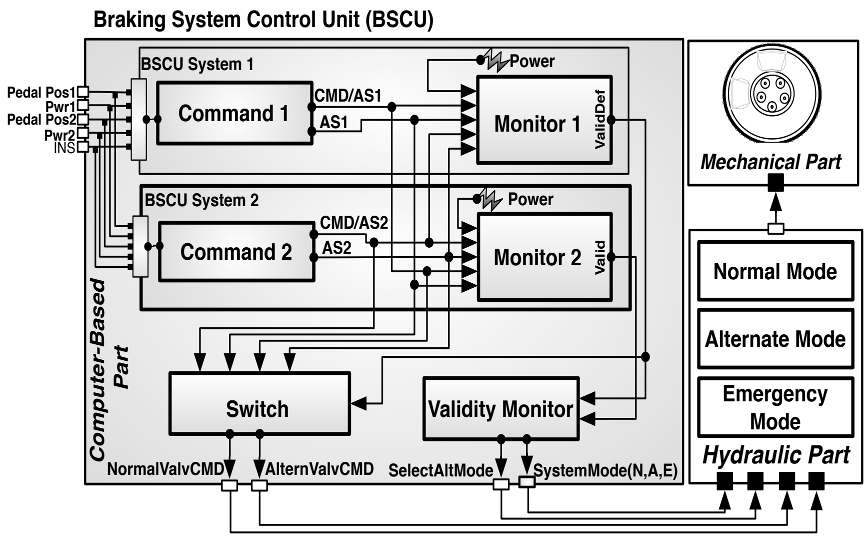


Figure 9.4: A high-level view of the WBS

to a the *Validity Monitor*. A failure reported by either system in the *BSCU* will cause that system to disable its outputs and set the *Validity Monitor* to invalid with no effect on the mode of operation of the whole system. However, if both monitors report failure the *BSCU* is deemed inoperable and is shut down [13].

It worth noting that Figure 9.4 shows high-level view of the *BSCU* implementation and it omits many details. However, the figure is still sufficient to illustrate key elements of our technique. More details about the *BSCU* implementation can be found in ARP-4761 [12].

9.4.2 Applying the Technique

Before we can apply the technique, both the required and actual failure probabilities of the top event should be clearly defined, where $FP_{Required(Topevent)} > FP_{Actual(Topevent)}$. Appendix L of the ARP-4761 states, as a safety requirement on the *BSCU*, that: *"The probability of BSCU fault causes Loss of Braking Commands shall be less than 3.30E-5 per flight"*. This means that: $FP_{Required(Topevent)} <$ **3.30E-5**. In line with this, we assumed that the $FP_{Actual(Topevent)} \approx$ **1.50E-6**. Figure 9.5 shows the "Loss of Braking

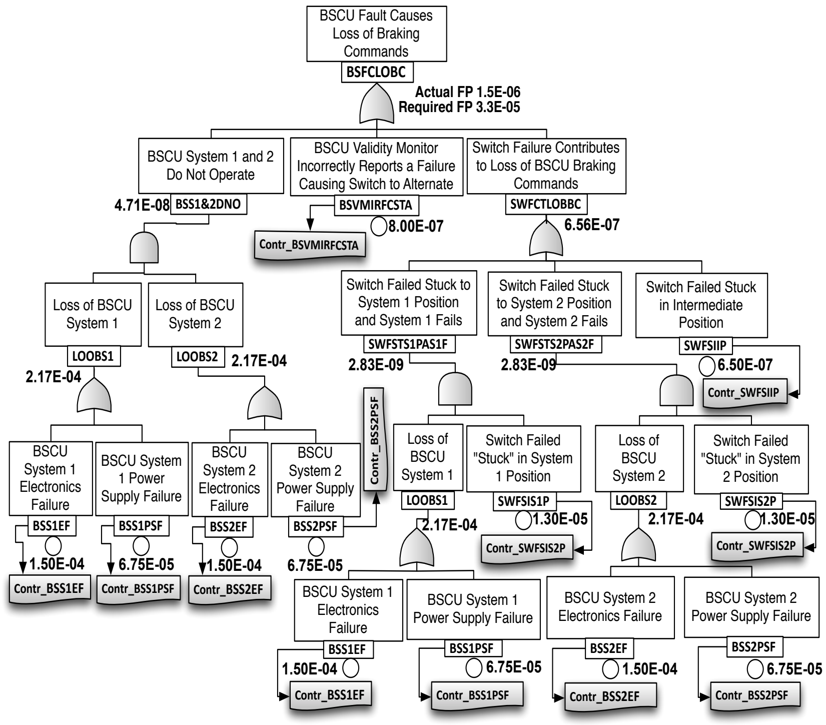

BSCU Fault Causes Loss of Braking Commands
BSFCLOBC
Actual FP 1.5E-06
Required FP 3.3E-05
BSCU System 1 and 2 Do Not Operate
BSS1&2DNO
4.71E-08
BSCU Validity Monitor Incorrectly Reports a Failure Causing Switch to Alternate
BSVMIRFCSTA
8.00E-07
Contr_BSVMIRFCSTA
Switch Failure Contributes to Loss of BSCU Braking Commands
SWFCTLOBBC
6.56E-07
Loss of BSCU System 1
LOOBS1
2.17E-04
Loss of BSCU System 2
LOOBS2
2.17E-04
Switch Failed Stuck to System 1 Position and System 1 Fails
SWFSTS1PAS1F
2.83E-09
Switch Failed Stuck to System 2 Position and System 2 Fails
SWFSTS2PAS2F
2.83E-09
Switch Failed Stuck in Intermediate Position
SWFSIIP
6.50E-07
Contr_SWFSIIP
BSCU System 1 Electronics Failure
BSS1EF
1.50E-04
Contr_BSS1EF
BSCU System 1 Power Supply Failure
BSS1PSF
6.75E-05
Contr_BSS1PSF
BSCU System 2 Electronics Failure
BSS2EF
1.50E-04
Contr_BSS2EF
BSCU System 2 Power Supply Failure
BSS2PSF
6.75E-05
Contr_BSS2PSF
Switch Failed "Stuck" in System 1 Position
SWFSIS1P
1.30E-05
Contr_SWFSIS2P
Switch Failed "Stuck" in System 2 Position
SWFSIS2P
1.30E-05
Contr_SWFSIS2P

Figure 9.5: Loss of Braking Commands FTA

Commands" probabilistic FTA.

- ***Step 1. Apply the sensitivity analysis to the "Loss of Braking Commands" probabilistic FTA***: the following steps were performed to apply the sensitivity analysis:

    1. Find minimal cut set MC in the FTA: there are several algorithms to find the MC. We apply Mocus cut set algorithm [11], as follows:

       $\mathbf{MC} = \{\mathbf{BSVMIRFCSTA} + \mathbf{SWFSIIP} + (\mathbf{BSS1EF} * \mathbf{BSS2EF}) + (\mathbf{BSS1EF} * \mathbf{BSS2PSF}) + (\mathbf{BSS1EF} * \mathbf{SWFSIS1P}) + (\mathbf{BSS1PSF} * \mathbf{BSS2EF}) + (\mathbf{BSS1PSF} * \mathbf{BSS2PSF}) + (\mathbf{BSS1PSF} * \mathbf{SWFSIS1P}) + (\mathbf{BSS2EF} * \mathbf{SWFSIS2P}) + (\mathbf{BSS2PSF} * \mathbf{SWFSIS2P})\}$.

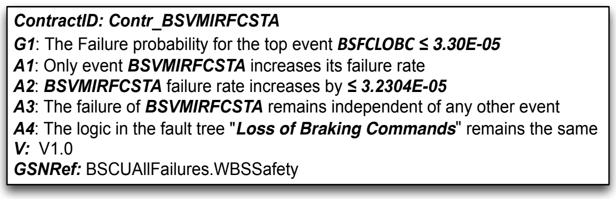
Table 9.1: The results of the sensitivity analysis

Event	$FP_{Actual\|event(x)}$	$\approx \Delta FP$	$FP_{Increased\|event(x)}$	***Sensitivity***	***Rank***
BSVMIRFCSTA	8.00E-07	3.150E-05	3.2304E-05	39	1
SWFSIIP	6.50E-07	3.150E-05	3.2154E-05	48	2
SWFSIS1P	1.30E-05	1.448E-01	1.4484E-01	51182	5
SWFSIS2P	1.30E-05	1.448E-01	1.4484E-01	51182	5
BSS1EF	1.50E-04	1.448E-01	1.4498E-01	965	3
BSS1PSF	6.75E-05	1.448E-01	1.4490E-01	2145	4
BSS2EF	1.50E-04	1.448E-01	1.4498E-01	965	3
BSS2PSF	6.75E-05	1.448E-01	1.4490E-01	2145	4

2. A simple C program was coded to calculate the maximum possible failure probability $FP_{Increased|event(x)}$ for each event in the MC. Subsequently, the $FP_{Actual|event(x)}$ was subtracted from the $FP_{Increased|event(x)}$ to obtain ΔFP for each event. Table 9.1 shows the calculated $FP_{Increased|event(x)}$ and ΔFP.
3. Applying the sensitivity equation: $(FP_{Increased|event(x)} - FP_{Actual|event(x)})/FP_{Actual|event(x)}$ determines the sensitivity for x where $x \in MC$. Table 9.1 shows the sensitivity values and the ranking, where 1 indicates the most sensitive event.

- ***Step 2. Refine the identified sensitive parts with system developers***: the WBS is a hypothetical system and no discussions have been made with the system developers. For the sake of the example, however, a pessimistic decision was made to consider all the events in Table 9.1 as liable to change. It is worth noting that in more complex examples the volume of sensitive event lists will be quite big. Hence, discussing those lists with system developers can lead to more selective events and thus alleviating the number of safety contracts.

- ***Step 3. Derive safety contracts from "Loss of Braking Commands" FTA***: based on the list of the sensitive events from Step 2, a safety contract was derived for each event in the list. The introduced safety contract template in Figure 9.3a was used to demonstrate the derived safety contracts. For lack of space, we show only one example of the eight derived safety contracts (see Figure 9.6).



ContractID: Contr_BSVMIRFCSTA
G1: The Failure probability for the top event ***BSFCLOBC ≤ 3.30E-05***
A1: Only event ***BSVMIRFCSTA*** increases its failure rate
A2: ***BSVMIRFCSTA*** failure rate increases by ***≤ 3.2304E-05***
A3: The failure of ***BSVMIRFCSTA*** remains independent of any other event
A4: The logic in the fault tree "***Loss of Braking Commands***" remains the same
V: V1.0
GSNRef: BSCUAllFailures.WBSSafety

Figure 9.6: A derived safety contract

- ***Step 4. Build the safety argument for the BSCU and associate the derived contracts with it***: a safety argument fragment was built as shown in Figure 9.7. The derived safety contracts are associated with the derived safety contracts according to ***Steps 4*** in Section 9.3. *BSCUAllFailures* claims that *BSCU* faults cause Loss of Braking commands are sufficiently managed. The possible faults of *BSCU*, based on “Loss of Braking Commands” FTA, are addressed by the subgoals below the *ArgAllCaus* strategy. Hence, *BSCUAllFailures* represents the top event of the FTA and thus the derived safety contracts are associated with it. The single black star on the left refers to the notation that is used to associate the contracts with *BSCUAllFailures*. It is important to note that goals in the gray color background represent assumptions in the safety contract. Each goal of those has the same name of the event in the assumptions list of the corresponding contract. For instance, *BSVMIRFCSTA* is a goal that represents an assumption within *Contr_BSVMIRFCSTA* contract which, in turn, guarantees a property for another event.

  The double black stars in the lower right corner refer to that annotation which is described in Section 9.3. It is important to make sure that contracts, related artefacts and items evidence have the same version number. The main idea of having the information within this notation is to pave the way to highlight the impact of changes. However, this idea will be described for the last three steps of the technique which is left for future work.

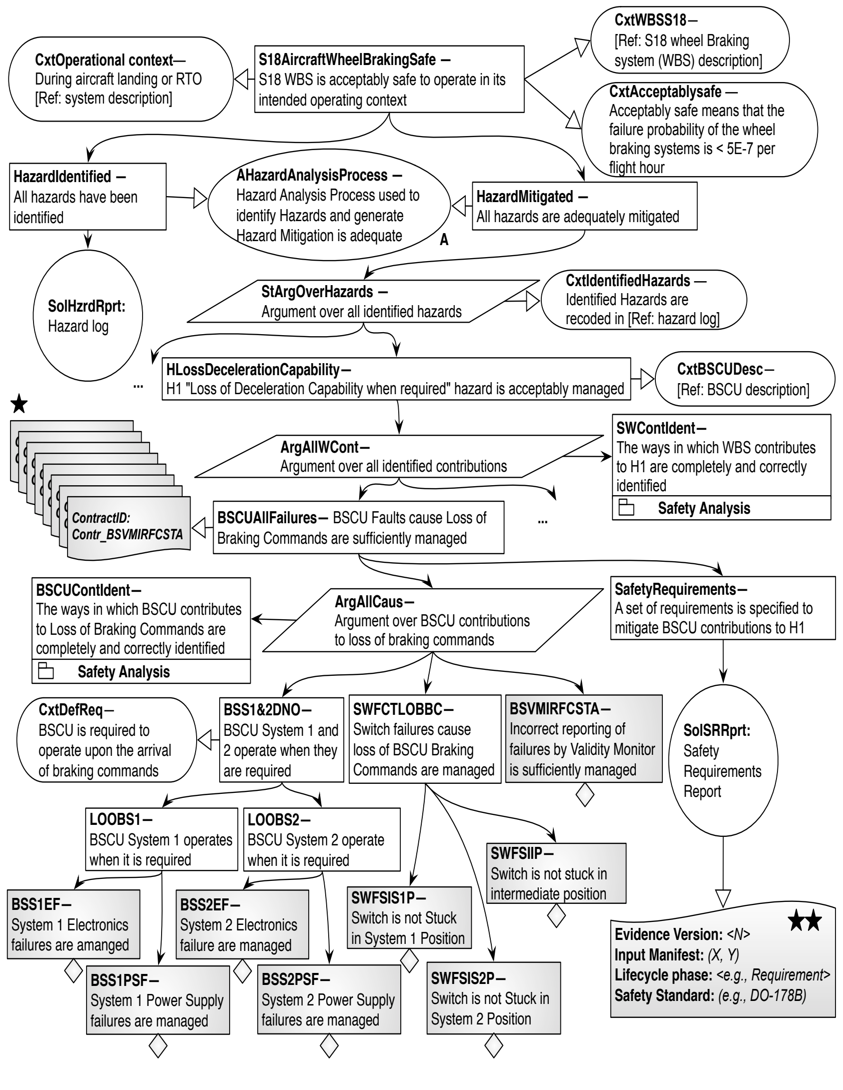

CxtWBSS18– [Ref: S18 wheel Braking system (WBS) description]
CxtOperational context– During aircraft landing or RTO [Ref: system description]
S18AircraftWheelBrakingSafe – S18 WBS is acceptably safe to operate in its intended operating context
CxtAcceptablysafe – Acceptably safe means that the failure probability of the wheel braking systems is < 5E-7 per flight hour
HazardIdentified – All hazards have been identified
AHazardAnalysisProcess – Hazard Analysis Process used to identify Hazards and generate Hazard Mitigation is adequate
A
HazardMitigated – All hazards are adequately mitigated
SolHzrdRprt: Hazard log
StArgOverHazards – Argument over all identified hazards
CxtIdentifiedHazards – Identified Hazards are recoded in [Ref: hazard log]
...
HLossDecelerationCapability– H1 "Loss of Deceleration Capability when required" hazard is acceptably managed
CxtBSCUDesc– [Ref: BSCU description]
★
ContractID: Contr_BSVMIRFCSTA
ArgAllWCont– Argument over all identified contributions
SWContIdent– The ways in which WBS contributes to H1 are completely and correctly identified
Safety Analysis
BSCUAllFailures– BSCU Faults cause Loss of Braking Commands are sufficiently managed
...
BSCUContIdent– The ways in which BSCU contributes to Loss of Braking Commands are completely and correctly identified
Safety Analysis
ArgAllCaus– Argument over BSCU contributions to loss of braking commands
SafetyRequirements– A set of requirements is specified to mitigate BSCU contributions to H1
CxtDefReq– BSCU is required to operate upon the arrival of braking commands
BSS1&2DNO– BSCU System 1 and 2 operate when they are required
SWFCTLOBBC– Switch failures cause loss of BSCU Braking Commands are managed
BSVMIRFCSTA– Incorrect reporting of failures by Validity Monitor is sufficiently managed
SolSRRprt: Safety Requirements Report
LOOBS1– BSCU System 1 operates when it is required
LOOBS2– BSCU System 2 operate when it is required
SWFSIIP– Switch is not stuck in intermediate position
BSS1EF– System 1 Electronics failures are amanged
BSS2EF– System 2 Electronics failure are managed
SWFSIS1P– Switch is not Stuck in System 1 Position
★★
Evidence Version: <N>
Input Manifest: (X, Y)
Lifecycle phase: <e.g., Requirement>
Safety Standard: (e.g., DO-178B)
BSS1PSF– System 1 Power Supply failures are managed
BSS2PSF– System 2 Power Supply failures are managed
SWFSIS2P– Switch is not Stuck in System 2 Position

Figure 9.7: Safety argument fragment for WBS

9.5 Related Work

A consortium of researchers and industrial practitioners called the Industrial Avionics Working Group (IAWG) has proposed using modular safety cases as a means of containing the cost of change. IAWG's Modular Software Safety Case (MSSC) process facilitates handling system changes as a series of relatively small increments rather than occasional major updates. The process proposes to divide the system into a set of blocks [3][4]. Each block may correspond to one or more software components but it is associated to exactly one dedicated safety case module. Engineers attempt to scope blocks so that anticipated changes will be contained within argument module boundaries. The process establishes component traceability between system blocks and their safety argument modules using Dependency-Guarantee Relationships (DGRs) and Dependency-Guarantee Contracts (DGCs). Part of the MSSC process is to understand the impact of change so that this can be used as part of producing an appropriate argument. The MSSC process, however, does not give details of how to do this. The work in this paper addresses this issue.

Kelly [14] suggests identifying preventative measures that can be taken when constructing the safety case to limit or reduce the propagation of changes through a safety case expressed in goal-structure terms. For instance, developers can use broad goals (goals that are expressed in terms of a safety margin) so that the these goals might act as barriers to the propagation of change as they permit a range of possible solutions. A safety case therefore, interspersed with such goals at strategic positions in the goal structure could effectively contain "firewalls" to change. Some of these initial ideas concerning change and maintenance of safety cases have been presented in [15]. However, no work was provided to show how these thoughts can facilitate the maintenance of safety cases.

9.6 Conclusion and Future Work

Changes are often only performed years after the initial design of the system making it hard for the designers performing the changes to know which parts of the argument are affected. Using contracts to manage system changes is not a novel idea any more since there has been significant work discusses how to represent contracts and how to use them. However, there has been little work on how to derive them. In this paper, we proposed a technique in which we showed a way to derive safety contracts using the sensitivity analysis. We also proposed a way to map the derived safety contracts to a safety argument to improve the change impact analysis on the safety argument and eventually facilitate its maintenance. Future work will focus on describing the last three steps of the technique. Also, creating a case study to validate both the feasibility and efficacy of the technique is part of our future work.

Acknowledgment

We acknowledge the Swedish Foundation for Strategic Research (SSF) SYN-OPSIS Project for supporting this work. We thank Patrick Graydon for his help and fruitful discussions of this paper.

Bibliography

[1] ISO 26262:2011. Road Vehicles — Functional Safety, Part 1-9. International Organization for Standardization, Nov 2011.

[2] GSN Community Standard: (http://www.goalstructuringnotation.info/) Version 1; (c) 2011 Origin Consulting (York) Limited.

[3] Modular software safety case (MSSC) — process description. https://www.amsderisc.com/related-programmes/, Nov 2012.

[4] Jane L Fenn, Richard D Hawkins, PJ Williams, Tim P Kelly, Michael G Banner, and Y Oakshott. The who, where, how, why and when of modular and incremental certification. In *Proceedings of the 2nd IET International Conference on System Safety*, pages 135–140. IET, 2007.

[5] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.

[6] L. Benvenuti, A. Ferrari, E. Mazzi, and A. L. Vincentelli. Contract-based design for computation and verification of a closed-loop hybrid system. In *Proceedings of the 11th International Workshop on Hybrid Systems: Computation and Control*, HSCC '08, pages 58–71, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] T.P. Kelly and J.A. McDermid. A systematic approach to safety case maintenance. In *Computer Safety, Reliability and Security*, volume 1698, pages 13–26. Springer Berlin Heidelberg, 1999.

[8] A.C. Cullen and H.C. Frey. *Probabilistic techniques in Exposure assessment*. Plenum Press, New York, 1999.

[9] David J. Pannell. Sensitivity analysis of normative economic models: theoretical framework and practical strategies. *Agricultural Economics*, 16(2):139 – 152, 1997.

[10] Omar Jaradat, Patrick J. Graydon, and Iain Bate. An approach to maintaining safety case evidence after a system change. In *Proceedings of the 10th European Dependable Computing Conference (EDCC)*, August 2014.

[11] Marvin Rausand and Arnljot Høyland. *System Reliability Theory: Models, Statistical Methods and Applications*. Wiley-Interscience, Hoboken, NJ, 2004.

[12] SAE ARP4761 Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, December 1996.

[13] O. Lisagor, M. Pretzer, C. Seguin, D. J. Pumfrey, F. Iwu, and T. Peikenkamp. Towards safety analysis of highly integrated technologically heterogeneous systems – a domain-based approach for modelling system failure logic. In *The 24th International System Safety Conference (ISSC)*, Albuquerque, USA, 2006.

[14] T. Kelly. Literature survey for work on evolvable safety cases. Department of Computer Sceince, University of York, 1st Year Qualifying Dissertation, 1995.

[15] S P Wilson, T P Kelly, and J A McDermid. Safety case development: Current practice, future prospects. In *proc. of Software Bases Systems - 12th Annual CSR Workshop*. Springer-Verlag, 1997.

Chapter 10

Paper E: Deriving Hierarchical Safety Contracts

Omar Jaradat and Iain Bate.

In Proceedings of the 21st IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2015), IEEE, Zhangjiajie, China, November 2015.

Abstract

Safety cases need significant amount of time and effort to produce. The required amount of time and effort can be dramatically increased due to system changes as safety cases should be maintained before they can be submitted for certification or re-certification. Sensitivity analysis is useful to measure the flexibility of the different system properties to changes. Furthermore, contracts have been proposed as a means for facilitating the change management process due to their ability to record the dependencies among system's components. In this paper, we extend a technique that uses a sensitivity analysis to derive safety contracts from Fault Tree Analyses (FTA) and uses these contracts to trace changes in the safety argument. The extension aims to enabling the derivation of hierarchical and correlated safety contracts. We motivate the extension through an illustrative example within which we identify limitations of the technique and discuss potential solutions to these limitations.

10.1 Introduction

The concept of a safety case originated in 1989 when the UK Health and Safety Executive (HSE) requested from the British nuclear sites to generate a written report — according to the Control of the Industrial Major Accident Hazards (CIMAH) regulations — that should contain: (1) facts about the site, and (2) reasoned arguments about the hazards and risks from the site [1]. This report is known as a safety case, which should systematically demonstrate a reasoned argument that a nuclear site is acceptably safe to operate. More specifically, a safety case should identify the major hazards and risks in a nuclear site and demonstrate that the site is satisfactory since all of these hazards and risks are adequately mitigated. The increasing size and complexity of safety critical systems motivate the application of the safety case concept in different domains (e.g., oil, avionics, railway, automotive, etc.) to demonstrate a reasoned argument that those systems are acceptably safe to operate.

Safety certification is typically imposed by authorities as a censorship procedure to control the development of safety critical systems. The certification processes are based on an evaluation of whether the hazards associated with a system are mitigated to an acceptable level. Developers must provide evidence of safety to their regulators. Safety cases can explain how this evidence shows that the system is acceptably safe to operate. Hence, the development of safety cases has become common practice.

The certification process is amongst the most expensive and time-consuming tasks in the development of safety critical systems. A key reason behind that is the increasing complexity and size of these systems combined with their growing market demands. Moreover, safety critical systems are expected to operate for a long period of time and frequently subject to changes during both development and operational phases. Any change that might compromise system safety involves repeating the certification process (i.e., re-certification) and thus, ultimately, necessitates maintaining the corresponding safety case. For example, the UK Ministry of Defence *Ship Safety Management System Handbook JSP 430* requires that “The safety case will be updated ... to reflect changes in the design and/or operational usage which impact on safety, or to address newly identified hazards. The safety case will be a management tool for controlling safety through life including design and operation role changes” [2, 3]. Similarly, the UK HSE *Railway safety case regulations 1994* states in regulation *6(1)* that “A safety case is to be revised whenever, appropriate that is whenever any of its contents would otherwise become inaccurate or incomplete” [4, 3].

One of the biggest challenges that affects safety case revision and maintenance is that a safety case comprises a complex web of interdependent elements. That is, safety goals, evidence, argument, and assumptions about operating context are highly interdependent and thus, seemingly minor changes may have a major impact on the contents and structure of the safety argument. As such, a single change to a safety case may necessitate many other consequential changes — creating a ripple effect [5]. For example, if a new system component was integrated into a system, old items of evidence might no longer support the developers' claims about components' consistency because these claims reflect old assumptions in the development artefacts that do not take into consideration the new component integration.

In order to maintain a safety case after implementing a system change, system developers need to assess the impact of changes on the original safety argument. The assessment shall include reviewing the relevant assumptions made in the argument, and examining the adequacy of the collected body of evidence. For example, the UK Defence *Standard DS 00-56* states that: "Any amendments to the deployment of the system should be examined against the assumptions and objectives contained in the safety case" [6], [3]. Hence, a step to assess the impact of this change on the safety argument is crucial and highly needed prior to updating a safety argument after a system change. Despite clear recommendations to adequately maintain and review safety cases by safety standards, existing standards offer little advice on how such operations can be carried out [5]. If developers do not understand the impact of change then they have to be conservative and do wider verification (i.e., check more elements than strictly necessary). This increases the maintenance cost.

Modularity has been proposed as the key element of the 'way forward' in developing systems [7]. For modular systems, the required maintenance efforts to accommodate predicted changes can be less than the required efforts to accommodate arbitrary changes. This is because having a list of predicted changes during the system design phase allows system engineers to contain the impact of each of those changes in a minimal number of system's modules. Hence, predicted changes can have traceable consequences as engineers will be aware of how a change in one module can result in a change in another module. In practice, it is hard to align the safety case structure with the system's modules [8]. However, a well-established traceability between system's modules and its safety case can provide the same traceable consequences of changes in the safety case. The problem though is that system changes and their details cannot be fully predicted and made available up front. In particular, software aspects of the safety case is hard to be predicted as software is highly changeable and harder to contain. In this paper, we use sensitivity analysis based approach to assist system's engineers to predict changes.

In our previous work [8], we introduced a technique that contains Sensitivity ANalysis for Enabling Safety Argument Maintenance (SANESAM) phase that supports system engineers to anticipate potential changes. The key principle of SANESAM phase is to determine the flexibility (or compliance) of a system to changes using sensitivity analysis. The output is a ranked list of FTA events that system engineers can refine. The result after the refinement is a list of contracts that can be used as part of later change impact analysis. We also use safety contracts to record the information of changes that will ultimately advise the engineers what to consider and check when changes actually happen. The main contribution of this paper comprises (1) identifying possible limitations for SANESAM, and (2) suggest an a SANESAM extension to resolve the identified limitations. The paper uses the hypothetical aircraft Wheel Braking System (WBS) to illustrate SANESAM extension.

This paper is composed of four further sections. In Section 10.2 we present background information about sensitivity analysis, safety contracts, Goal Structuring Notations (GSN), incremental certification and WBS description. In Section 10.3, we give an overview of a technique to facilitate the maintenance of safety cases and identify limitations. In Section 10.4, we suggest extending the technique to resolve the identified limitations, we also use the WBS system to illustrate the extensions. Finally, we conclude and propose future works in Section 10.5.

10.2 Background

10.2.1 Sensitivity Analysis

Sensitivity analysis can be defined as: "The study of how uncertainty in the output of a model (numerical or otherwise) can be apportioned to different sources of uncertainty in the model input" [9]. The analysis helps to establish reasonably acceptable confidence in the model by studying the uncertainties that are often associated with variables in models [10]. There are different purposes for using sensitivity analysis, such as, providing insight into the robustness of model results when making decisions [11]. For instance, sensitivity analysis can be used to determine what level of accuracy is necessary for a parameter (variable) to make the model sufficiently useful and valid [12].The analysis can be also used to enhance communication from modelers to decision makers, for example, by making recommendations more credible, understandable, compelling or persuasive [13]. The analysis can be performed by different methods, such as, mathematical, graphical, statistical, etc.

Emberson et al. [14] use sensitivity analysis to improve the flexibility of task allocation in real-time system design. More specifically, the analysis is used to evaluate the impact on task allocation solution after applying possible change scenarios to task allocation framework.

In this paper, we apply the sensitivity analysis on FTAs to measure the sensitivity of outcome A (e.g., a safety requirement being true) to a change in a parameter B (e.g., the failure probability in a component). The sensitivity is defined as $\Delta B/B$, where ΔB is the smallest change in B that changes A (e.g., the smallest increase in failure probability that makes safety requirement A false). The failure probability values that are attached to FTA's events are considered input parameters to the sensitivity analysis. A sensitive part of a FTA is defined as one or multiple FTA events whose minimum changes (i.e., the smallest increase in its failure probability due to a system change) have the maximal effect on the FTA, where effect means exceeding failure probabilities (reliability targets) to inadmissible levels. A sensitive event is an event whose failure probability value can significantly influence the validity of the FTA once it increases. A sensitive part of a FTA is assigned to a system design component that is referred to as a sensitive component in this paper. Changes to a sensitive component cause a great impact to system design. [8]

10.2.2 Safety Contracts

The concept of contract is familiar in software development. For instance, Design by Contract (DbC) was introduced in 1986 [15, 16] to constrain the interactions that occur between objects. Contract-based design is an approach where the design process is seen as a successive assembly of components where a component behaviour is represented in terms of assumptions about its environment and guarantees about its behavior [17]. In the context of contract-based design, a contract is conceived as an extension to the specification of software component interfaces that specifies preconditions and postconditions to describe what properties a component can offer once the surrounding environment satisfies one or more related assumption(s). A contract is said to be a *safety contract* if it guarantees a property that is traceable to a hazard. There have been significant works that discuss how to represent and to use contracts [18, 19, 20]. In the safety critical systems domain, researchers have used, for example, assume-guarantee contracts to propose techniques to lower the cost of developing software for safety critical systems. Moreover, contracts have been exploited as a means for helping to manage system changes in a system domain or in its corresponding safety case [21, 22, 23].

The following is an example that depicts the most common used form of contracts:

Guarantee: The WCET of task X is $\leq$ 10 milliseconds
Assumptions:
X is:

1. compiled using compiler $[C]$,
2. executed on microcontroller $[M]$ at 1000 MHz with caches disabled, and
3. not interrupted

10.2.3 Safety Argumentation and Goal Structuring Notations (GSN)

GSN was introduced to provide a graphical means of communicating (1) safety argument elements, claims (goals), argument logic (strategies), assumptions, context, evidence (solutions), and (2) the relationships between these elements [24]. The principal symbols of the notations (with example instances of each concept) are shown in Figure 10.1. A goal structure shows how goals are successively broken down into ('solved by') sub-goals until a point is reached where claims can be supported by direct reference to evidence. Using GSN, the writer can clarify the adopted argument strategies (i.e., how the premises imply the conclusion), the rationale for the approach (assumptions, justifications) and the context in which goals are stated [8]. GSN has been extended to enable modularity in safety cases (i.e., module-based development of the safety case) so that it enables the partitioning of a safety case into an interconnected set of modules.

Well-structured argument may help the developers to mechanically propagate the change through the goal structure. However, it does not tell if the suspect elements of the argument in question are still valid. For example, having made a change to a model we must ask whether goals articulated over that model are still valid. Expert judgment is still required in order to answer such questions [25]. Hence, merely having well-structured arguments does not directly help to preserve the soundness of the argument after a change, but it can more easily determine the questions to be asked to do so.

10.2.4 Incremental Certification

The Industrial Avionics Working Group (IAWG) — a consortium of researchers and practitioners — has proposed Modular Safety Cases as a means of containing the cost of change by dividing the safety case into a set of argument modules. IAWG's Modular Software Safety Case (MSSC) process [26] facilitates handling system changes as a series of relatively small increments rather than occasional major updates. The key principle of the state-of-the-art process is to modularise a safety case so as to contain changes within a minimal area of the safety case [26]. More specifically, the process starts by anticipating potential changes over the lifetime of a system. System developers modularise the argument so as to contain the impact of the anticipated changes. Hence, MSSC process ensures that the maximum amount of safety case material that was previously certified is not impacted, and thus it is available for re-use in the re-certification process without a need to be revisited [26].

10.2.5 Wheel Braking System (WBS): System Description

The WBS is a hypothetical aircraft braking system described in Appendix L of a popular standard for safety assessment processes, ARP4761 [27]. Figure 10.2 shows a high-level architecture view of the WBS. The system is installed on the two main landing gears of a civil air transport. The main function of the system is to provide wheel braking as commanded by the pilot when the aircraft is on

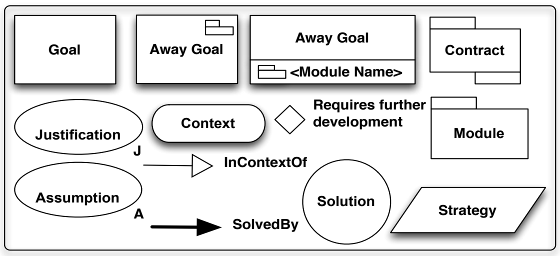


Figure 10.1: Overview of GSN Notations

the ground. The system is composed of three main parts: 1. Computer-based part which is called the Brake System Control Unit *(BSCU)*, 2. *Hydraulic* part, and 3. *Mechanical* part.

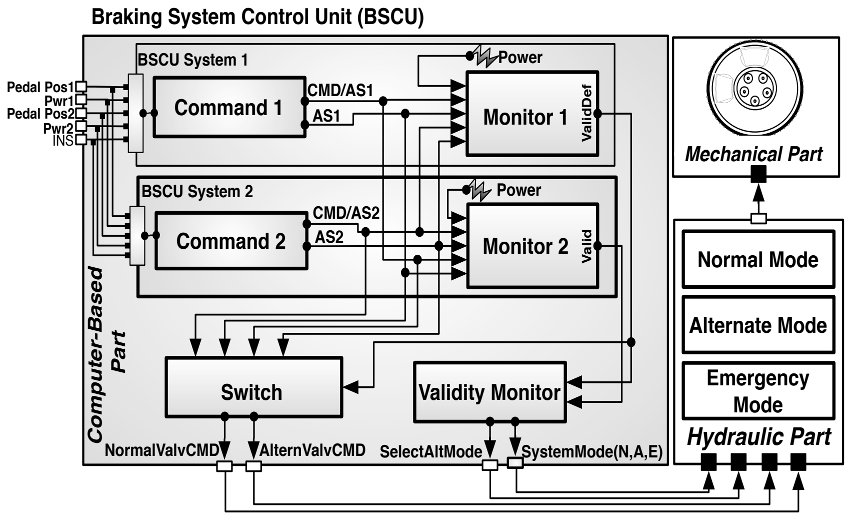


Figure 10.2: A high-level view of the WBS [8]

The *BSCU* is internally redundant and consists of two channels, *BSCU System 1 and 2* (*BSCU* is the box in the gray background in Figure 10.2). Each channel consists of two components: *Monitor* and *Command*. *BSCU System 1 and 2* receive the same pedal position inputs, and both calculate the command value. The two command values are individually monitored by the *Monitor 1 and 2*. Subsequently, values are compared and if they do not agree, a failure is reported. The results of both *Monitors* and the compared values are provided to a the *Validity Monitor*. A failure reported by either system in the *BSCU* will cause that system to disable its outputs and set the *Validity Monitor* to invalid with no effect on the mode of operation of the whole system. However, if both monitors report failure, the *BSCU* is deemed inoperable and is shut down [8, 27, 28]. Figure 10.2 shows high-level view of the *BSCU* implementation and it omits many details. However, the figure is still sufficient to illustrate key elements of our technique. More details about the *BSCU* implementation can be found in ARP-4761 [27]. Figure 10.3 shows the "Loss of Braking Commands" probabilistic FTA.

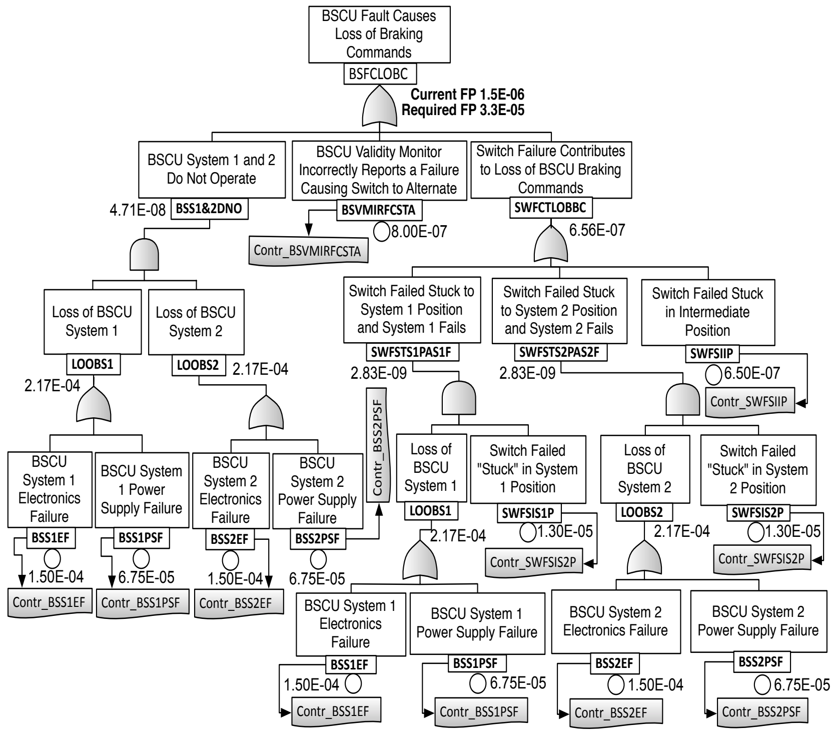

BSCU Fault Causes Loss of Braking Commands
BSFCLOBC
Current FP 1.5E-06
Required FP 3.3E-05
BSCU System 1 and 2 Do Not Operate
BSS1&2DNO
4.71E-08
BSCU Validity Monitor Incorrectly Reports a Failure Causing Switch to Alternate
BSVMIRFCSTA
8.00E-07
Contr_BSVMIRFCSTA
Switch Failure Contributes to Loss of BSCU Braking Commands
SWFCTLOBBC
6.56E-07
Loss of BSCU System 1
LOOBS1
2.17E-04
Loss of BSCU System 2
LOOBS2
2.17E-04
Switch Failed Stuck to System 1 Position and System 1 Fails
SWFSTS1PAS1F
2.83E-09
Switch Failed Stuck to System 2 Position and System 2 Fails
SWFSTS2PAS2F
2.83E-09
Switch Failed Stuck in Intermediate Position
SWFSIIP
6.50E-07
Contr_SWFSIIP
BSCU System 1 Electronics Failure
BSS1EF
1.50E-04
Contr_BSS1EF
BSCU System 1 Power Supply Failure
BSS1PSF
6.75E-05
Contr_BSS1PSF
BSCU System 2 Electronics Failure
BSS2EF
1.50E-04
Contr_BSS2EF
BSCU System 2 Power Supply Failure
BSS2PSF
6.75E-05
Contr_BSS2PSF
Loss of BSCU System 1
LOOBS1
2.17E-04
Switch Failed "Stuck" in System 1 Position
SWFSIS1P
1.30E-05
Contr_SWFSIS2P
Loss of BSCU System 2
LOOBS2
2.17E-04
Switch Failed "Stuck" in System 2 Position
SWFSIS2P
1.30E-05
Contr_SWFSIS2P
BSCU System 1 Electronics Failure
BSS1EF
1.50E-04
Contr_BSS1EF
BSCU System 1 Power Supply Failure
BSS1PSF
6.75E-05
Contr_BSS1PSF
BSCU System 2 Electronics Failure
BSS2EF
1.50E-04
Contr_BSS2EF
BSCU System 2 Power Supply Failure
BSS2PSF
6.75E-05
Contr_BSS2PSF

Figure 10.3: Loss of Braking Commands FTA [8]

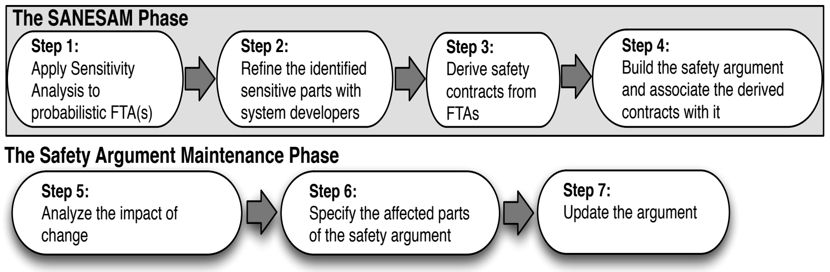

The SANESAM Phase
Step 1: Apply Sensitivity Analysis to probabilistic FTA(s)
Step 2: Refine the identified sensitive parts with system developers
Step 3: Derive safety contracts from FTAs
Step 4: Build the safety argument and associate the derived contracts with it
The Safety Argument Maintenance Phase
Step 5: Analyze the impact of change
Step 6: Specify the affected parts of the safety argument
Step 7: Update the argument

Figure 10.4: Process diagram of safety cases maintenance technique

10.3 A Technique to Facilitate the Maintenance of Safety Cases

In this section we give an overview of a technique that aims to facilitate the maintenance of a safety case. The technique comprises 7 steps that are distributed between the Sensitivity ANalysis for Enabling Safety Argument Maintenance (SANESAM) phase and the safety argument maintenance phases as shown in Figure 10.4. The steps of the SANESAM phase are represented along the upper path, whilst the lower path represents the steps of the safety argument maintenance phase. Considering a complete list of anticipated changes is difficult. This technique uses sensitivity analysis to measure the flexibility of system components to changes. That is, regardless of the type of changes it will be seen as factors to increase or decrease a certain parameter value. Thus system developers can focus more on predicting those changes that might make the parameter value inadmissible [8]. Furthermore, the technique utilises the concept of contracts to record the information of changes that will ultimately advise the engineers what to consider and check when changes actually happen.

10.3.1 SANESAM Phase

The rationale of this phase is to determine, for each component, the allowed range for a certain parameter within which a component may change before it compromises a certain system property (e.g., safety, reliability, etc.). Sensitivity analysis is used in this phase as a method to determine the range of failure probability parameter for each component. The technique assumes the existence of a probabilistic FTA where each event in the tree is specified by a current estimate of failure probability $FP_{Current|event(x)}$. In addition, the technique assumes the existence of the required failure probability for the top event $FP_{Required(Topevent)}$, where the FTA is considered unreliable if: $FP_{Currentl(Topevent)} > FP_{Required(Topevent)}$. [8]

The steps of SANESAM phase are as follows: [8]

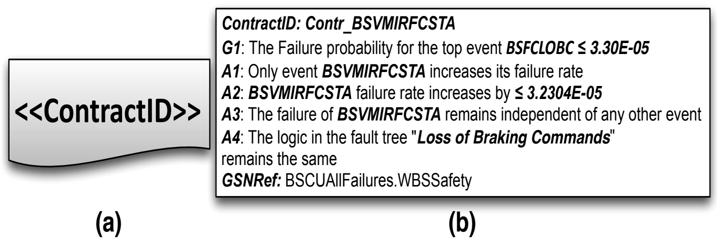
- ***Step 1. Apply the sensitivity analysis to a probabilistic FTA***: In this step the sensitivity analysis is applied to a FTA to identify the sensitive events whose minimum changes have the maximal effect on the $FP_{Topevent}$. Identifying those sensitive events requires the following steps to be performed:

1. Find the Minimal Cut Set (MC) in the FTA. The minimal cut set definition is: *"A cut set in a fault tree is a set of basic events whose (simultaneous) occurrence ensures that the top event occurs. A cut set is said to be minimal if the set cannot be reduced without losing its status as a cut set"* [29].
2. Calculate the maximum possible increment in the failure probability parameter of event x before the top event $FP_{Required(Topevent)}$ is no longer met, where $x \in MC$ and $(FP_{Increased|event(x)} - FP_{Current|event(x)}) \nRightarrow FP_{Increased(Topevent)} > FP_{Required(Topevent)}.$
3. Rank the sensitive events from the most sensitive to the less sensitive. The most sensitive event is the event for which the following formula is the minimum:

$$\frac{FP_{Increased|event(x)} - FP_{Current|event(x)}}{FP_{Current|event(x)}.}$$

- ***Step 2. Refine the identified sensitive parts with system developers***: In this step, the generated list of sensitive events from Step 1 should be discussed by system developers (e.g., safety engineers) as they should choose the sensitive events that are most likely to change. The list can be extended to add any additional events by the developers. Moreover, it is envisaged that some events might be removed from the list or the rank of some of them might change.

- ***Step 3. Derive safety contracts from FTAs***: In this step, a safety contract or contracts should be derived for each event in the list from Step 2. The main objectives of the contracts are to 1) highlight the sensitive events to make them visible up front for developers attention, and 2) to record the dependencies between the sensitive events and the other events in the FTA. Hence, if the system is later changed in a way that increases the failure probability of a contracted event where the increased failure probability is still within the defined threshold in the contract, then it can be said that the contract(s) in question still hold (intact) and the change is containable with no further maintenance. The contract(s), however, should be updated to the latest failure probability value. On the other hand, if the change causes a bigger increment in the failure probability value than the contract can hold, then the contract is said to be broken and the guaranteed event will no longer meet its reliability target. It is



ContractID: Contr_BSVMIRFCSTA
G1: The Failure probability for the top event ***BSFCLOBC ≤ 3.30E-05***
A1: Only event ***BSVMIRFCSTA*** increases its failure rate
A2: ***BSVMIRFCSTA*** failure rate increases by ***≤ 3.2304E-05***
A3: The failure of ***BSVMIRFCSTA*** remains independent of any other event
A4: The logic in the fault tree "***Loss of Braking Commands***" remains the same
GSNRef: BSCUAllFailures.WBSSafety

Figure 10.5: (a) FTA Safety contract notation, (b) Derived safety contract

worth noting that the role of safety contracts in SANESAM is to highlight sensitive events, and not to enter new event failure probabilities. We introduce a new notation to FTAs to annotate the contracted events, where every created contract should have a unique identifier, see Figure 10.5-a. Figure 10.3 shows the derived safety contracts as a result of SANESAM application to the Loss of Braking Command FTA. We also create a template to document the derived safety contracts. Figure 10.5-b shows an instantiation of the contents of one of the derived safety contracts for WBS.

- ***Step 4. Build the safety argument and associate the derived contracts with it***: In this step, a safety argument should be built and the derived safety contracts should be associated with the argument elements. Associating the contracts with GSN goals is done by using the introduced notation in Figure 10.5-a.

10.3.2 SANESAM Limitations

The essence of SANESAM is to calculate the maximum possible increment in the failure probability parameter of only one event at a time before the top event $FP_{Required(Topevent)}$ is no longer met. In this section, we identify three limitations of the current version of SANESAM and we give an example for each limitation.

No Support for Intermediate Events

The followed method for applying sensitivity analysis relies on the calculated cut set for the full FTA. Hence, only basic events are considered during the application of sensitivity analysis and no contracts are derived for the intermediate events. Figure 10.6-a shows an example of this limitation, where LOOBS1 is an intermediate event and based on SANESAM it cannot be provided as a sensitive event thus no contract can be derived for it. Having said that, system developers may need to contain the impact of changes in intermediate levels to prevent them from rippling through the top-level event. Additionally, some events might look trivial for system engineers but if those events were packed in events from higher levels, then they could look nontrivial. For example, providing system engineers with “Jam of speedbrake lever” as a sensitive event to a particular change might look less serious than the parent event “Mechanical failures of speedbrake lever”. Another motivation for deriving contracts on intermediate levels comes from the fact that some intermediate events may represent top goals in the safety case modules which will be more supportive for incremental certification as introduced in Section 10.2.4. In other words, pinpointing the entire safety case module as affected is easier than starting from intermediate goals in that module. From the forgoing reasons, SANESAM should be able to provide system engineers with sensitive events from different levels of the FTA’s hierarchy.

No Support for Multiple Events Impact

SANESAM calculates the highest possible boundary of failure probabilities for certain events. SANESAM also assumes independence of events and does not address the problem of event interdependencies that is typical for any realistic system. This means that only one event failure probability is allowed to increase per change. However, a change might impact multiple events in the same time. For instance, adding distinct functional redundancy of a critical software component might decrease the failure probability of multiple events in the FTA. Likewise, removing a redundant component to make the system simpler and cheaper might increase the failure probability of multiple events. Since the failure probabilities of multiple components often change at once, a SAMESAM extension to handle such changes is highly desirable. A clear example can be given by assuming a change to BSCU System 1 power supply in Figure 10.6-a. A change to BSCU System 1 power supply will necessitate a correlated change to BSCU System 2 power supply. Hence, when we need to calculate the possible increment to the failure probability of BSS1PSF (for this specific change), we must take into account the correlation between BSS1PSF and BSS2PSF.

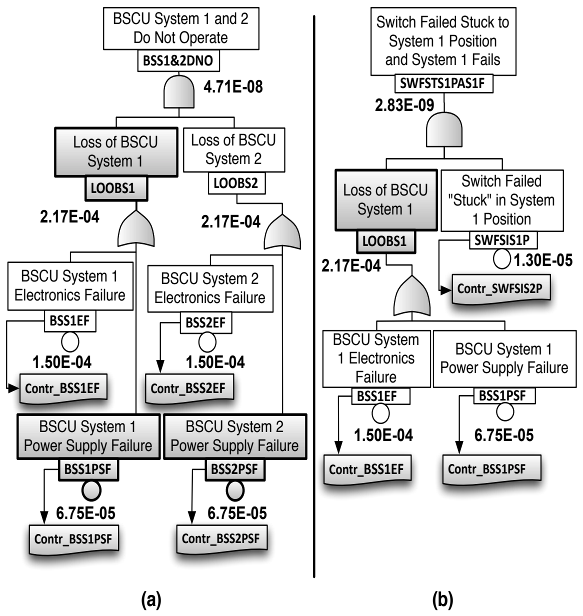


Figure 10.6: Limitation examples

Neglecting Duplication

It is possible for an event to be represented in more than one location in the same FTA. For example, LOOBS1 event is represented twice in this FTA. In the first representation (Figure 10.6-a) it is combined with LOOBS2 by AND gate, where both events have the same failure probability. In its second representation (Figure 10.6-b), LOOBS1 is combined with SWFSIS1P by AND gate, both events have different failure probabilities. Hence, the possible increment on LOOBS1 failure probability will vary in the two locations. SANESAM neglects events duplication, and this is considered a limitation because the calculated possible failure probability increment of the same event may vary in the same FTA if the event is duplicated. Calculating the possible increment on the failure probability of a duplicated event is based on the failure probability(s) of the combined events. More clearly, in each duplication of an event, the event may be combined with different event(s), different failure probability values, and by different gates (e.g., AND, OR, XOR, etc.).

10.4 SANESAM Extension

In this section, we suggest extending SANESAM to resolve the limitations that were identified in Section 10.3.2. The extended SANESAM is referred to as SANESAM+ in this paper. SANESAM and SANESAM+ are mutually exclusive and selecting among them is very dependent on the refined list by system engineers in Step 2 of the technique (described in Section 10.3.1). More clearly, if at least one of the events in the refined list is duplicated, or if its change necessitates a correlated event to change, then SANESAM+ is the one to go. Otherwise, developers are free to choose SANESAM or SANESAM+. However, choosing SANESAM means that the developers accept the assumption that only one event is allowed to change at a time.

SANESAM relies on the calculated MC (minimum cut set) for the full FTA which means that only the basic events are considered for sensitivity analysis. However, SANESAM+ requires measuring the sensitivity of all events in the FTA. This means that we need to calculate the Maximum Allowed Failure Probability (MAFP) for each event in the FTA taking into account that all events may change at a time. That is, $\Delta FP_{(Topevent)}$ = ($FP_{Required(Topevent)}$ $- FP_{Current(Topevent).}$) will be distributed over all FTA's events, where $\Delta FP_{(Topevent)} > 0$.
In order to apply SANESAM+ and calculate the MAFP for FTA events, we replace the procedure of Step 1 in Section 10.3.1 with the following procedure:

1. Find the difference between new and current FPs of the ancestor events, as follows:

$$\Delta FP_{(Ancestor)} = FP_{New} - FP_{Current}$$

   The first run of this step should start with $\Delta FP_{(Topevent)}$, where the new FP in this specific case is the required FP. The second run should be for each event in the very next level and so on and so forth until the basic events are reached.

2. This sub-step is very dependent on the type of the gate between the ancestor and descendant events. In case of OR gate, sub-steps 2-A and 2-B should be followed. In case of AND gate, sub-step 2-C should be followed.

   (a) Find the ratio of the descendant events to the ancestor event. The first run of this step should start with the top event and the events beneath it. The second run of this step should consider one more level down. In other words, descendant events in the first run will become ancestors in the second one. The ratio of a descendant event to its ancestor is calculated by Equation 10.1, as follows:

$$Ratio_{Desc(x)} = \frac{FP_{Current(Desc(x))}}{FP_{Current(Ancestor)}} \tag{10.1}$$

(b) Increase the FP for each of the descendant events by $\Delta FP_{(Ancestor)}$ which is calculated in step 1. Increasing events' FP is done by Equation 10.2, as follows:

$$\begin{aligned} FP_{Increased|Desc(x)} &= FP_{Current(Desc(x))} \\ &+(Ratio_{(Desc(x))} * \Delta FP_{(Ancestor)}) \end{aligned} \tag{10.2}$$

(c) In this sub-step, we need to distribute the increment to the FP of an ancestor event over its descendent events in the presence of an AND gate. The increment to each descendant event is calculated in two different ways based on the number of descendent events and if their FPs vary.

Case 1. if the events share the same FP value, we can use: $\sqrt[n]{FP_{(Increased|Ancestor)}}$, where n is the number of the descendent events.

LOOBS1 and LOOBS2 in Figure 10.6-a represents an example of this case.

Case 2. if the descendent events do not share the same FP, then $FP_{(Topevent)}$ is distributed over them unevenly, but rather based on the FP ratio of every descendent event to $\Delta FP_{(Ancestor)}$ as described by Equation 10.3:

$$FP_{Current(Desc(x))} + (\frac{FP_{Current(Desc(x))}}{\sum FP_{Current(AllDesc)}} * I) \tag{10.3}$$

In order to determine I we need to consider all sibling events as described in Equation 10.4:

$$\begin{aligned}
&(FP_{Current(Desc(x_1))} + (\frac{FP_{Current(Desc(x_1))}}{\sum FP_{Current(AllDesc)}} * I)) \\
&*(FP_{Current(Desc(x_2))} + (\frac{FP_{Current(Desc(x_2))}}{\sum FP_{Current(AllDesc)}} * I)) \\
&*(FP_{Current(Desc(x_n)} + (\frac{FP_{Current(Desc(x_n))}}{\sum FP_{Current(AllDesc)}} * I)) \\
&\qquad = FP_{Increased(Ancestor)}
\end{aligned} \tag{10.4}$$

LOOBS1 and SWFSIS1P in Figure 10.6-b represent an example of this case.

3. Repeat steps 1 and 2 until FP of the basic events get increased. Unlike SANESAM, SANESAM+ distinguish between duplicated events. That is, if an event shows up in multiple locations in the FTA, we still need to calculate its FP wherever we encounter it. Later on when finish calculating the FP for all duplicates of an event we unify the its FP by considering the minimum calculated FP of them.

4. Finally, rank the sensitivity of events from the most sensitive to the less sensitive. The most sensitive event is the event for which Equation 10.5 is the minimum, as follows:

$$Sensitivity_{(x)} = \frac{FP_{Increased(x)} - FP_{Current(x)}}{FP_{Current(x)}} \tag{10.5}$$

It is worth noting that the difference between the steps of SANESAM and SANESAM+ is observed only in Step 1, all other later steps are identical.

10.4.1 SANESAM+ Application: An Example

In this section, we use the Loss of Braking Commands FTA (in Figure 10.3) to show an application example of SANESAM+.

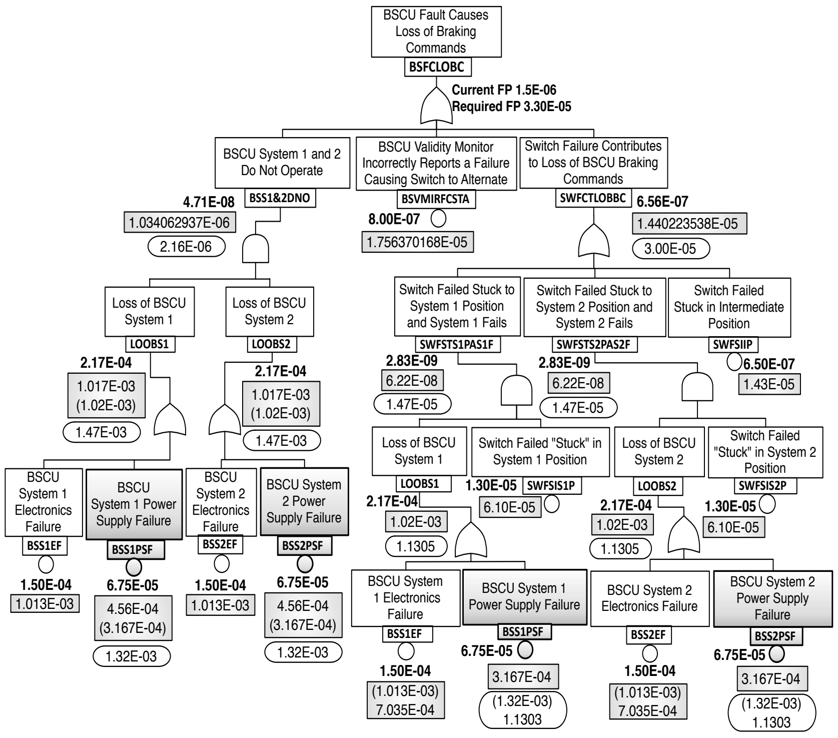


Figure 10.7: Loss of Braking Commands FTA [8]

1. Find $\Delta FP_{(Ancestor)}$ (which is the top event in the FTA for the first of this sub-step): $\Delta\ FP_{(BSFCLOBC)}$ = 3.30E-05 - 1.5031E-06
$\Delta\ FP_{(BSFCLOBC)}$ = 3.14969E-05

2. Since BSFCLOBC is correlated with its descendants (i.e., BSS1&2DNO, BSVMIRFCSTA and SWFCTLOBBC) via OR gate, then sub-steps 2-A and 2-B should be followed.

   (a) We need to find FP ratio for each of BSS1&2DNO, BSVMIRFC-STA and SWFCTLOBBC to BSFCLOBC using Equation 10.1.

   Example: BSS1&2DNO

$$Ratio_{BSS1\&2DNO} = \frac{4.71E\text{–}08}{1.5031E\text{–}06}$$

$Ratio_{BSS1\&2DNO}$ = 3.133524050E-02.

(b) In this sub-step, $\Delta FP_{(BSFCLOBC)}$ should be distributed over each of BSS1&2DNO, BSVMIRFCSTA and SWFCTLOBBC based on their ratios to $FP_{Actaul(Ancestor)}$ using Equation 10.2.

Example: BSS1&2DNO

$FP_{Increased(BSS1\&2DNO)}$ =
4.71E-08 + (3.133524050E-02 * 3.14969E-05)

$FP_{Increased(BSS1\&2DNO)}$ = 1.034062937E-06

(c) Now, let us take other examples where AND gate correlates an ancestor event with its descendent events. The example covers Case 1 and 2 as described in sub-step 2-C in Section 10.4.

Example of Case 1: LOOBS1 and LOOB2.

$FP_{(Increased(x)}$ = $\sqrt[2]{FP_{(Increased|BSS1\&2DNO)}}$
$FP_{(Increased|LOOBS1)}$ = 1.016889E-03
$FP_{(Increased|LOOBS2)}$ = 1.016889E-03

Example of Case 2: LOOBS1 and SWFSIS1P
In this example, the FP of LOOBS1 and SWFSIS1P are increased using Equations 10.3 and 10.4.

$$= (2.17E\text{–}04 + (\frac{2.17E\text{–}04}{2.17E\text{–}04 + 1.30E\text{–}05} * I)) *$$
$$(1.30E\text{–}05 + (\frac{1.30E\text{–}05}{1.30E\text{–}05 + 2.17E\text{–}04} * I))$$
$$= 6.216381375E\text{–}08$$

$FP_{(Increased|LOOBS1)}$ = 1.02E-03
$FP_{(Increased|SWFSIS1P)}$ = 6.10E-05

3. In this step, we repeat the 1 and 2 steps until FP of the basic events get increased. Figure 10.7 shows the calculated FPs (in boxes) using SANESAM+. Looking at the figure, It should be observed that the events, BSS1EF, BSS2EF, BSS1PSF and BSS2PSF are duplicated.

These events were assumed independent from each other while calculating their FPs. However, this assumption was vanished after the calculation and the minimum FP (values between brackets in Figure 10.7) was considered the maximum possible FP for each duplicate events. For example, two FP values were obtained for BSS1EF in different locations (i.e., 4.56E-04 and 3.167E-04) but since 3.167E-04 is the minimum FP value of the two duplicates, it is, therefore, the maximum possible FP for all BSS1EF's duplicates.

4. In this step, we use Equation 10.5 in Section 10.4. Table 10.1 shows the results of the sensitivity analysis and the ranking of the events' sensitivity where 1 is the most sensitive event.

Table 10.1: The results of SANESAM+ sensitivity analysis

Event Name	Current FP	Increased FP	Sensitivity	Rank
SWFSIS1P **SWFSIS2P**	1.30E-05	6.10E-05	3.692307692	2
LOOBS1 **LOOBS2**	2.17E-04	1.02E-03	3.700460829	3
BSS1EF **BSS2EF**	1.50E-04	1.013E-03	5.753333333	4
BSS1PSF **BSS2PSF**	6.75E-05	3.167E-04	3.69185185	1
SWFCTLOBBC	6.56E-07	1.44E-05	20.95121951	5
SWFSTS1PAS1F **SWFSTS2PAS2F**	2.83E-09	6.22E-08	20.97879859	6
BSVMIRFCSTA	8.00E-07	1.76E-05	21	7
SWFSIIP	6.50E-07	1.43E-05	21	7
BSS1&2DNO	4.71E-08	1.04E-06	21.08067941	8

10.4.2 SANESAM+ For Predicted Changes

SANESAM+ can be useful even for arbitrary changes. That is, even if the system engineers are not sure of the potential future changes, SANESAM+ enable the derivation of safety contracts for all events in different levels in the FTA. Hence, when a change request shows up, system engineers, and by returning to the sensitivity results, can decide whether the effect of the change is tolerable or not. However, SANESAM+ can be more useful in the presence of a predicted change as it can increase the effect tolerance of that change. More clearly, distributing $\Delta FP_{(Topevent)}$ over all FTA's events might increase the change impact tolerance of some events that are unlikely to change. On the other hand, the change impact tolerance might be slightly increased for events that are more likely to change. Consequently, having a change scenario in advance will motivate increasing the change impact tolerance for only the events that fall in the scope of that change. Since, however, SANESAM+ (for predicted changes) will exclude the events that are unlikely to change, we will slightly modify the steps by which we calculate the FP of events. The following steps give guidance on how to calculate the FP SANESAM+ for predicted change scenarios:

1. Find the difference between the current and required FP of the top event $\Delta FP_{(Topevent)}$.

2. Find the highest event that contains the effect. If the highest event does not fall directly under the top event, the effect should be traced up the fault tree further until we reach the affected event that falls directly under the top event.

3. Distribute the calculated $\Delta FP_{(Topevent)}$ in sub-step 1 to the identified events in sub-step 2 based on the determined ratio in sub-step 3. The first run of this sub-step should start with the top event and the events beneath it, and the second runshould consider one more level down.This sub-step is very dependent on the type of the gate between the ancestor and descendant events. In the case of an OR gate sub-step 4-A should be followed. In the case of an AND gate sub-step 4-B should be followed.

   (a) In this sub-step, we need to distribute the increment to the FP of an ancestor event over its descendent affected events in the presence of an OR gate. We first need to find the ratio of the affected event to its ancestor event. Afterwards, we need to use the calculated ratio to determine the amount of the increment to the affected event.

The first run of this step should start with the affected events that fall directly under the top event. The second run of this step should consider one more level down. In other words, descendant events in the first run will become ancestors in the second one. The simplest FP calculation is when to have two descendent events and only one of them is affected. This is because all what we need to do is to subtract the unaffected FP from the increased ancestor event to get the the increased FP of the affected event as presented in Equation 10.6:

$$\begin{split} FP_{Increased(Ancestor)} - FP_{Current(Unaffect|Desc(x))} \\ = FP_{Increased(Desc(x))} \end{split} \tag{10.6}$$

Otherwise, the ratio of a descendant event to its ancestor and the granted increment to an affected event is calculated by Equation 10.7 as follows:

$$\begin{split} FP_{Increased(Desc(x))} = \\ (\frac{FP_{Current(Desc(x))}}{FP_{Current(Ancestor)} - \sum FP_{Current(Unaffect)}} \\ * FP_{Increased(Ancestor)}) + FP_{Current(x)} \end{split} \tag{10.7}$$

(b) In this sub-step, we distribute the increment to the FP of an ancestor event over its affected descendent events in the presence of an AND gate. The increment calculation is dependent on five cases, as follows:
Case 1. Two descendent events and only one of them is affected. This is the simplest case because all what we need to do is to divide the increased FP of the ancestor event on the current FP of the unaffected descendent event as presented in Equation 10.8:

$$FP_{Increased(Desc(x))} = \frac{FP_{Increased(Ancestor)}}{FP_{Current(Unaffect|Desc(x))}} \tag{10.8}$$

Case 2. All descendent events are affected and share the same FP value. In this case, we apply:
$\sqrt[n]{FP_{(Increased|Ancestor)}}$, where n is the number of the descendent events.
Case 3. All descendent events are affected and do **NOT** share the same FP value. In this case, we apply equations (3) and (4) as described in Section 10.4.

Case 4. NOT all descendent events are affected where the affected ones share the same FP value. In this case, we apply Equation 10.9 as follows:

$$FP_{Increased(Desc(x))} = \sqrt[n]{(\frac{FP_{Increased(Ancestor(x))}}{\sum FP_{Current|Unaffect(x_1)*(x_2)*...*(x_n))}}} \tag{10.9}$$

where n is the number of the affected events.

Case 5. NOT all descendent events are affected where the affected ones do **NOT** share the same FP value. In this case, we use Equation 10.10, as follows:

$$\frac{FP_{Increased(Ancestor(x))}}{\sum FP_{Current|Unaffected(x_1)*(x_2)*...*(x_n))}} \tag{10.10}$$

4. Repeat step 3 until the FP of all affected events get increased.

10.4.3 SANESAM+ For Predicted Changes: An Example

In this example we again use the WBS FTA. We consider a predicted change that will be applied to the power supplies within both $BSCU1$ and 2. However, it is still unknown how this change will increase the FPs of the two power supplies. We apply "SANESAM+ For Predicted Changes" to dedicate the maximum allowed FP to the affected events by the change, as follows:

1. $\Delta\ FP_{(BSFCLOBC)}$ = 3.30E-05 - 1.5031E-06
   $\Delta\ FP_{(BSFCLOBC)}$ = 3.14969E-05

2. Find the highest event that contains the effect. Changes to System 1 and 2 power supplies will directly affect the events BSS1EF and BSS2EF as highlighted in Figure 10.7 . These two events, however, are duplicated elsewhere in the FTA and thus there are multiple high events that contain the change.

   (a) **BSS1EF on the left-hand side of the FTA** falls under LOOBS1 but the latter does not fall directly under the top event thus BSS1&2DNO is the highest event that contains the effect on BSS1EF.

   (b) **BSS2EF on the left-hand side of the FTA** falls under LOOBS2 but the latter does not fall directly under the top event thus BSS1&2DNO is the highest event that contains the effect on BSS2EF.

(c) **BSS1EF on the right-hand side of the FTA** falls under LOOBS1 but the latter does not fall directly under the top event thus it is not the required highest event and we need to take one more level up to find the highest event. Having done that will lead us to SWFSTS1PAS1F which is also not the highest event that falls directly under the top event thus we need to go up again which will result SWFCTLOBBC as the required highest event.

(d) **BSS2EF on the right-hand side of the FTA** falls under LOOBS2 but the latter does not fall directly under the top event thus it is not the required highest event and we need to take one more level up to find the highest event. Having done that will lead us to SWFSTS1PAS2F which is also not the highest event that falls directly under the top event thus we need to go up again which will result SWFCTLOBBC as the required highest event.

3. Distribute the increment to the FP of an ancestor event over its descendent affected events. Since BSS1&2DNO and SWFCTLOBBC are the events that contain the change, no further calculations will be applied to BSVMIRFCSTA.

$$(\frac{4.71E\text{–}08}{1.5031E\text{–}06 - 8.00E\text{–}07} * 3.30E\text{–}05) + 4.71E\text{–}08$$

$$= 2.16E\text{–}06$$

$$(\frac{6.56E\text{–}07}{1.5031E\text{–}06 - 8.00E\text{–}07} * 3.30E\text{–}05) + 6.56E\text{–}07$$

$$= 3.00E\text{–}05$$

4. In this step, we repeat the previous step until all FPs of the affected events get increased. Figure 10.7 shows the calculated FPs (in squashed boxes).

5. In this step, we use 10.5 to calculate events' sensitivity.

It is worth noting that the sensitivity of BSS1PSF and BSS2PSF using SANESAM+ is 3.69185185 as shown in Table 10.1. However, the sensitivity of these events is 18.55555556 when SANESAM+ For Predicted Changes is used.

10.5 Conclusions and Future Work

Sensitivity analysis is useful to measure the flexibility of different system properties to changes. In our previous work, we proposed a technique comprises of two phases to facilitate the maintenance of safety cases. SANESAM is the first phase of the technique in which we (1) measure the sensitivity of FTA events to system changes using the events' failure probabilities, and (2) derive safety contracts based on the results of the analysis. In the second phase, we map the derived safety contracts to a safety argument to improve the change impact analysis on the safety argument. In this paper, we identified some limitations to SANESAM and we suggested two options as extensions to resolve these limitations. The first option is SANESAM+, which is useful in the case of arbitrary changes because it calculates the FP for all events in the FTA regardless of any change scenario. The second option is SANESAM+ For Predicted Changes, this option increases the FP for only the events that are associated to a predicted change. A derived safety contract by SANESAM+ For Predicted Changes can guarantee higher FP than the guaranteed FP (for the same event and using the same set of assumptions) in a derived safety contract by SANESAM+. Hence, the derived safety contracts by SANESAM+ For Predicted Changes are more tolerant and robust than those derived by SANESAM+. Future work will focus on describing the second phase of the technique. Creating a case study to validate both the feasibility and efficacy of the technique is also part of our future work.

Acknowledgment

We acknowledge the Swedish Foundation for Strategic Research (SSF) SYNOPSIS Project for supporting this work. We thank Patrick Graydon for his help and fruitful discussions of this paper.

Bibliography

[1] CASSIDY K. CIMAH safety cases - the HSE approach. IChemE Symposium series no. 110, 1988.

[2] U.K. Ministry of Defence, “JSP 430 - Ship Safety Management System Handbook,” Ministry of Defence January 1996.

[3] Timothy Patrick Kelly. Arguing safety – a systematic approach to managing safety cases, 1998.

[4] HSE, ”Railway Safety Cases - Railway (Safety Case) Regulations 1994 - Guidance on Regulations,” Health and Safety Executive, HSE Books 1994.

[5] T.P. Kelly and J.A. McDermid. A systematic approach to safety case maintenance. In *Proceedings of the Computer Safety, Reliability and Security*, volume 1698 of *Lecture Notes in Computer Science*, pages 13–26. 1999.

[6] U.K. Ministry of Defence, 00-56 Safety Management Requirements for Defence Systems, Ministry of Defence, Defence Standard December 1996.

[7] S Bates, I Bate, R Hawkins, T P Kelly, J A McDermid, and R Fletcher. Safety case architectures to complement a contract-based approach to designing safe systems. In *Proceedings of the 21st International System Safety Conference (ISSC)*, 2003.

[8] Omar Jaradat, Iain Bate, and Sasikumar Punnekkat. Using sensitivity analysis to facilitate the maintenance of safety cases. In *Proceedings of the 20th International Conference on Reliable Software Technologies*, June 2015.

[9] A. Saltelli. *Global sensitivity analysis: the primer*. John Wiley, 2008.

[10] Omar Jaradat, Iain Bate, and Sasikumar Punnekkat. Facilitating the maintenance of safety cases. In *Proceedings of the 3rd International Conference on Reliability, Safety and Hazard - Advances in Reliability, Maintenance and Safety (ICRESH-ARMS)*, June 2015.

[11] A.C. Cullen and H.C. Frey. *Probabilistic techniques in Exposure assessment*. Plenum Press, New York, 1999.

[12] Mark Choudhari Lucia Breierova. An introduction to sensitivity analysis. Technical report, Massachusetts Institute of Technology, September 1996.

[13] David J. Pannell. Sensitivity analysis of normative economic models: theoretical framework and practical strategies. *Agricultural Economics*, 16(2):139 – 152, 1997.

[14] P. Emberson and I. Bate. Stressing search with scenarios for flexible solutions to real-time task allocation problems. *Software Engineering, IEEE Transactions on*, 36(5):704–718, Sept 2010.

[15] B. Meyer. Design by contract. Technical Report TR-EI-12/CO, Interactive Software Engineering Inc., 1986.

[16] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.

[17] L. Benvenuti, A. Ferrari, E. Mazzi, and A. L. Vincentelli. Contract-based design for computation and verification of a closed-loop hybrid system. In *Proceedings of the 11th International Workshop on Hybrid Systems: Computation and Control*, pages 58–71, Berlin, Heidelberg, 2008. Springer-Verlag.

[18] Albert Benveniste, Benoit Caillaud, Alberto Ferrari, Leonardo Mangeruca, Roberto Passerone, and Christos Sofronis. Multiple viewpoint contract-based specification and design. In *Proceedings of the 6th International Symposium*, pages 200–225, October 2007.

[19] Werner Damm, Hardi Hungar, Bernhard Josko, Thomas Peikenkamp, and Ingo Stierand. Using contract-based component specifications for virtual integration testing and architecture design. In *Proceedings of the Design,*

Automation & Test in Europe Conference & Exhibition (DATE), pages 1–6, 2011.

[20] Sebastian S. Bauer, Alexandre David, Rolf Hennicker, Kim Guldstrand Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Moving from specifications to contracts in component-based design. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering*, FASE'12, pages 43–58, Berlin, Heidelberg, 2012. Springer-Verlag.

[21] Jane L Fenn, Richard D Hawkins, PJ Williams, Tim P Kelly, Michael G Banner, and Y Oakshott. The who, where, how, why and when of modular and incremental certification. In *Proceedings of the 2nd IET International Conference on System Safety*, pages 135–140. IET, 2007.

[22] Safecer. (2013, June) Safety certification of software-intensive systems with reusable components. [Online]. Available: http://www.safecer.eu.

[23] Patrick Graydon and Iain Bate. The nature and content of safety contracts: Challenges and suggestions for a way forward. In *Proceedings of the 20th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, November 2014.

[24] GSN Community Standard: (http://www.goalstructuringnotation.info/) Version 1; (c) 2011 Origin Consulting (York) Limited.

[25] S P Wilson, T P Kelly, and J A McDermid. Safety case development: Current practice, future prospects. In *Proceedings of the 12th Annual CSR Workshop - Software Based Systems*. Springer-Verlag, 1997.

[26] Modular Software Safety Case (MSSC) — Process Description. [online]. available: https://www.amsderisc.com/related-programmes, Nov 2012.

[27] SAE ARP4761 Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, December 1996.

[28] O. Lisagor, M. Pretzer, C. Seguin, D. J. Pumfrey, F. Iwu, and T. Peikenkamp. Towards safety analysis of highly integrated technologically heterogeneous systems – a domain-based approach for modelling system failure logic. In *Proceedings of the 24th International System Safety Conference (ISSC)*, Albuquerque, USA, 2006.

[29] Marvin Rausand and Arnljot Høyland. *System Reliability Theory: Models, Statistical Methods and Applications*. Wiley-Interscience, Hoboken, NJ, 2004.