Mälardalen University Press Licentiate Theses
No. 203

FACILITATING REUSE OF SAFETY CASE ARTEFACTS USING SAFETY CONTRACTS

Irfan Šljivo

2015

School of Innovation, Design and Engineering

ISBN 978-91-7485-213-4
ISSN 1651-9256
Printed by Arkitektkopia, Västerås, Sweden

Populärvetenskaplig sammanfattning

Säkerhetskritiska system är system som kan orsaka skada på egendom, miljö eller till och med mänskligt liv om de inte fungerar som de ska. Sådana system behöver vanligtvis följa en branschspecifik säkerhetsstandard som ofta innefattar en säkerhetsbevisning i form av argument för systemets funktionssäkerhet med tillhörande bevis att systemet säkert kan användas i avsedda sammanhang. Att utveckla säkerhetskritiska system så att de följer säkerhetsstandarder är en tidsödande och kostsam process. För att reducera kostnad och tid är återanvändning önskvärd. Ofta är säkerhetsbevisningen mer kostsam än utvecklingen av själva systemet. Därför behöver återanvändning av säkra komponenter i systemet kompletteras med återanvändning av delar av säkerhetsbevisningen.

Det svåra med att återanvända säkerhetsbevisning ligger i att säkerhet är en egenskap hos själva systemet som inte enkelt kan härledas från de ingående delarna. Det finns dessutom inte något systematiskt sätt att återanvända delar av en säkerhetsbevisning. Genom åren har många olyckor rapporterats vara en direkt konsekvens av osystematisk återanvändning. Till exempel så kraschlandade en Ariane 5 raket på grund av en mjukvarukomponent som återanvändes från den tidigare Ariane 4 versionen. Trots att Ariane 4 hade fullt fungerade och funktionssäker mjukvara så orsakade mjukvaran ett fel när den återanvänds i det nya systemet, vilket resulterade i kraschen som innebar miljardförluster.

I denna avhandling presenterar vi en form av säkerhetskontrakt som kan användas för att underlätta systematisk återanvändning av komponenter i säkerhetskritiska system. Ett säkerhetskontrakt för en komponent består av ett antagande och en garanti, sådant att komponenten erbjuder garantin givet att an-

tagandet uppfylls av den omgivning som komponenten används i. I avhandlingen undersöker vi följande i detalj: hur sådana kontrakt kan specificeras, hur de kan härledas, och på vilket sätt de kan användas för återanvändning i säkerhetsbevisning. Först kategoriserar vi kontrakten som antingen “starka” eller “svaga”, för att kunna fånga upp varianter av beteenden som återanvändbara komponenter kan påvisa i olika system. Sedan presenterar vi metoder för att härleda säkerhetskontrakt från felanalyser. Felanalys är en etablerad teknik för att identifiera risker i säkerhetskritiska system. Slutligen utvecklar vi metoder för att med hjälp av säkerhetskontrakt kunna återanvända delar av säkerhetsbevisning. Eftersom säkerhetsstandarder vanligen inte stöder systematisk återanvändning så definierar vi en process för utveckling och systematisk återanvändning av säkerhetskontrakt. Vi använder ett verkligt fall för att demonstrera hur våra metoder kan användas i enlighet med denna process.

Abstract

Safety-critical systems usually need to comply with a domain-specific safety standard, which often require a safety case in form of an explained argument supported by evidence to show that the system is acceptably safe to operate in a given context. Developing such systems to comply with a safety standard is a time-consuming and costly process. Reuse within development of such systems has a potential to reduce the cost and time needed to develop both the system and the accompanying safety case. Efficient reuse of safety-relevant components that constitute the system requires the reuse of the accompanying safety case artefacts, including the safety argument and the supporting evidence. The difficulties with reuse of the such artefacts within safety-critical systems lie mainly in the nature of safety being a system property, together with the lack of support for systematic reuse of such artefacts.

In this thesis we focus on developing a notion of safety contracts that can be used to facilitate systematic reuse of safety-relevant components and their accompanying artefacts. More specifically, we explore the following issues: in which way such contracts should be specified, how they can be derived, and in which way they can be utilised for reuse of safety artefacts. First, we characterise the contracts as either "strong" or "weak" to facilitate capturing different behaviours reusable components can exhibit in different contexts. Then, we present methods for deriving safety contracts from failure analyses. As the basis of the safety-critical systems development lies in the failure analyses and identifying which malfunctions could lead to accidents, the basis for specifying the safety contracts lies in capturing information identified by such failure analyses. Finally, we provide methods for generating safety case artefacts from safety contracts. Moreover, we define a safety contracts development process as guidance for systematic reuse based on the safety contracts. We use a real-world case to demonstrate the proposed process and methods.

To my family

Acknowledgments

First and foremost, I would like to express my immense gratitude to my supervisory team Hans Hansson, Jan Carlson and Barbara Gallina without whom this thesis would not be possible. Thank you for your invaluable guidance and endless patience you shared with me selflessly throughout these years.

I would like to thank to all those that have influenced my decision to pursue PhD studies and those that have made it possible, especially Hans Hansson and Sasikumar Punnekkat for accepting me as a PhD student. Special thanks goes to Damir Isović, Aida Čaušević, Adnan Čaušević and Zikrija Avdagić for their support of my decision to pursue PhD studies at Mlardalen University.

My deepest gratitude goes to my co-authors as well as the members of SYNOPSIS[1] research project for all the positive influence they had on my research. I am extremely grateful to Hans Hansson, Jan Carlson, Barbara Gallina, Patrick Graydon and Iain Bate for all the useful discussions and the vast knowledge they have shared with me. I would also like to thank Iain Bate, Patrick Graydon, Stefano Puri and Omar Jaradat as my co-authors outside of my supervisory team. A big thank you goes to other SYNOPSIS members especially Henrik Thane, Björn Lisper, Thomas Nolte, Kristina Lundqvist, Kaj Hänninen, Guillermo Rodriguez-Navas, Hüseyin Aysan, Husni Khanfar and Mahnaz Malekzadeh (Anita).

During my studies I have taken a number of courses. I wish to express my appreciation to all the lecturers and professors from whom I have learned how to be a better researcher. Many thanks to Ivica Crnković, Gordana Dodig-Crnković, Damir Isović, Jan Gustafsson, Iain Bate, Hans Hansson, Kristina Lundqvist, Cristina Seceleanu, Moris Behnam, Thomas Nolte, Emma Nehrenheim and Harold Lawson. A special thanks goes to the web team Malin R., Hüseyin, Leo and Predrag for making departmental duties fun. I would also

[1] http://www.es.mdh.se/SYNOPSIS/

like to thank the IDT administration staff for their support with practical issues. Many thanks to Carola, Susanne, Sofia, Malin(s), Ingrid, Anna, and the others.

Next, I wish to express my gratitude to all the great people I have met at our department with whom I have shared many joyful moments during our coffee, dessert and lunch breaks, sports activities, barbecues, conference and leisure trips, and all the other fun activities we did throughout the past years. A special thanks goes to my office room mates Omar, Gabriel, Husni and Anita for without them the light in our office would be rarely on.

Last but not least, I would like to thank my family, especially my parents Ajkuna and Kemal, my brother Faruk and his family, and my aunt Razija. Thank you for your endless love, inspiration and support you have given me.

The work in this thesis has been supported by the Swedish Foundation for Strategic Research (SSF) via project SYNOPSIS as well as EU and Vinnova via the Artemis JTI project SafeCer.

Irfan Šljivo
May, 2015
Västerås, Sweden

List of publications

Papers included in the licentiate thesis[2]

Paper A *Strong and Weak Contract Formalism for Thrid-Party Component Reuse*, Irfan Šljivo, Barbara Gallina, Jan Carlson, Hans Hansson. In Proceedings of the 3rd International Workshop on Software Certification (WoSoCer), IEEE, November 2013.

Paper B *Generation of Safety Case Argument-Fragments from Safety Contracts*, Irfan Šljivo, Barbara Gallina, Jan Carlson, Hans Hansson. In Proceedings of the 33rd International Conference on Computer Safety, Reliability, and Security (SafeComp), Springer-Verlag, September 2014.

Paper C *A Method to Generate Reusable Safety Case Fragments from Compositional Safety Analysis*, Irfan Šljivo, Barbara Gallina, Jan Carlson, Hans Hansson, Stefano Puri. In Proceedings of the 14th International Conference on Software Reuse (ICSR 2015), Springer-Verlag, January 2015.

Paper D *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, January 2015.

Paper E *Using Safety Contracts to Guide the Integration of Reusable Safety Elements within ISO 26262*, Irfan Šljivo, Barbara Gallina, Jan Carlson, Hans Hansson. Technical Report, ISSN 1404-3041, ISRN MDH-MRTC-300/2015-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, March 2015.

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

Additional papers, not included in the licentiate thesis

1. *Towards a Safety-oriented Process Line for Enabling Reuse in Safety Critical Systems Development and Certification*, Barbara Gallina, Irfan Šljivo, Omar Jaradat. In Proceedings of the 35th Annual Software Engineering Workshop (ISOLA workshop) (SEW 2012), IEEE, October 2012.

2. *Fostering Reuse within Safety-critical Component-based Systems through Fine-grained Contracts*, Irfan Šljivo, Jan Carlson, Barbara Gallina, Hans Hansson. In Proceedings of the 1st International Workshop on Critical Software Component Reusability and Certification across Domains (ICSR workshop) (CSC2013), June 2013.

3. *Facilitating Reuse of Certification Artefacts Using Safety Contracts*, Irfan Šljivo. In Proceedings of the Doctoral Symposium at the 14th International Conference on Software Reuse (ICSR 2015), January 2015.

Contents

I

Thesis

Chapter 1

Introduction and Outline

Safety-critical systems are those systems whose malfunctioning can result in harm or loss of human life, or damage to property or the environment [1]. A trend in safety-critical systems is that new functionalities are added mainly through software, which explains why a modern car has from 70 to 100 embedded computers on board, with overall software that scales up to 100 million lines of code[1]. To ensure that these safety-critical software-intensive systems achieve sufficient levels of safety, most of such systems must comply with a set of domain-specific safety standards. In this thesis we refer to the process of achieving compliance with a particular standard as certification process. The cost of achieving certification is estimated at 25-75% of the development costs [2], with the cost of producing the verification artefacts for highly critical applications reaching up to 1000 USD per code line [3].

In most cases, safety standards require a safety case to assure that any unreasonable residual risks due to the malfunctioning of the system and its elements have been avoided. A safety case is presented in the form of an explained and structured argument supported by evidence to clearly communicate that the system is acceptably safe to operate in a given context [4]. While the safety case includes all the artefacts (e.g., results of failure analyses or verification evidence) produced during the compliance process to assure that the system is acceptably safe, the safety argument represents means to connect the safety claims (e.g., that the system is acceptably safe to operate in a given context) with the safety case artefacts that provide supporting evidence (Figure 1.1).

More and more safety standards are offering support for reuse to reduce the

[1] see http://spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code

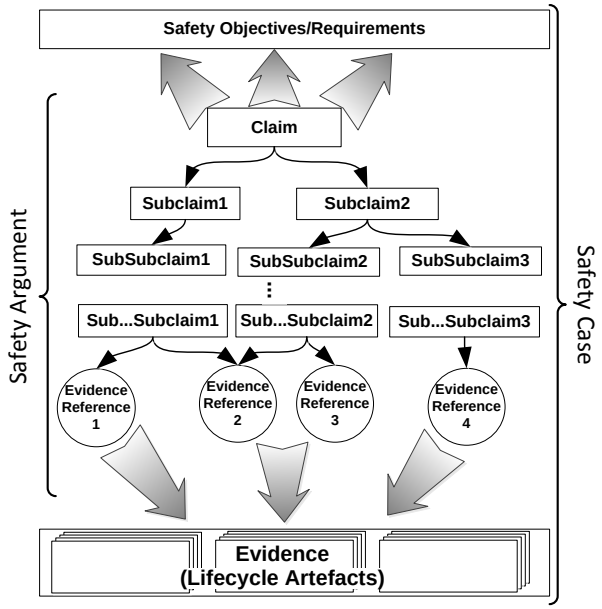

Figure 1.1: The role of safety argumentation within a safety case

production costs and time needed to achieve certification. For example, in the latest versions of both airborne (DO178-C) and automotive (ISO 26262) industry standards, techniques for reuse are explicitly supported through the notions of Reusable Software Components (RSC) within airborne [5], and Safety Elements out of Context (SEooC) within automotive industry [6]. Whether reuse is planned (systematic) or "ad hoc" (non-systematic) has significant influence on the safety of the system within which reuse is performed [7], hence safety standards typically take in consideration whether the safety element being reused is developed for reuse or not. For example, the SEooC concept within the ISO 26262 safety standard is used for elements that are developed for reuse and according to the standard, while for the other elements that are not necessarily developed for reuse nor according to ISO 26262, specific requirements for their qualification are defined. The planned reuse assumes that the elements being reused have been developed with reuse in mind, which usually results in higher development costs of the reusable element itself, but the return of investment can be achieved if the element is reused often enough [8]. Non-

systematic reuse usually does not incur additional development costs, but in return the level of reuse is minimal since reuse is done individually by reusing information in an "ad hoc" manner. Reuse within safety-critical systems comes in many different flavours. For example, we distinguish between the following reuse scenarios:

- Reuse of safety elements between system versions
- Reuse within a family of products (e.g., a product line)
- Cross-domain reuse (e.g., reuse of safety elements across the automotive and airborne domains where different safety standards apply)

Efficient reuse of safety-relevant components in either of these scenarios requires the reuse of the accompanying safety case artefacts, including the safety argument and the supporting evidence. The difficulty of achieving reuse of the safety case artefacts increases as the complexity of the scenarios increases (e.g., unlike in the first scenario, the cross-domain reuse scenario requires considering the compliance of the safety element and the accompanying artefacts with respect to the different domain-specific standards).

While safety standards provide requirements on what should be subject to reuse, guidance on performing systematic reuse of safety elements in the different reuse scenarios are not included in the standards, nor is there much guidance in the literature. This lack of support for systematic reuse of safety elements comes from the skepticism of the safety community in integrating and reusing elements developed without the real knowledge of the context in which the element will be used. This skepticism stems from the fact that safety is a system property, hence traditional failure analyses such as Fault Tree Analysis, as well as other safety case artefacts such as safety case arguments, are typically made at the system-level. Moreover, non-systematic reuse of safety artefacts has shown to be dangerous [7], hence there is a need to bridge the gap between the need of safety-critical industries for reuse and the skepticism of the safety community by establishing approaches for systematic reuse of safety elements.

Different approaches can be used to facilitate systematic software reuse. For example, Component-based Software Engineering (CBSE) is the most commonly used approach to achieve reuse within the airborne industry [9]. According to CBSE, software is developed by composing pre-existing or newly developed components, i.e., independent units of software, with a well-defined interface capturing communication and dependencies towards the rest of the

system [10]. As a part of CBSE approaches for safety-critical systems, contract-based approaches have received significant attention for some time already [11, 12, 13, 14]. A contract for a component is defined as an assumption/guarantee pair, where the component offers guarantees about its behaviour provided that the assumptions on its environment hold. A behaviour in this context is a sequence of values of a variable/property of the component and the environment. While similar CBSE approaches have been successfully used to support reuse of software components, they lack support for reuse of the accompanying safety artefacts.

Systematic reuse of safety case artefacts can be achieved by generating artefacts for a specific system from specifications written in a domain specification language, often referred to as generative reuse [15]. For example, a safety-relevant component developed out-of-context together with a safety argument is reused in a particular system. Since such argument could contain information that might be irrelevant for the particular system in which the component is reused, system-specific information should be captured in specifications so that system-specific safety argument could be generated for the particular system. In our work we focus on developing the notion of safety contracts that can capture such information and that can be used as a basis for an approach to systematic reuse of safety element and the accompanying safety artefacts.

In this thesis we address the following three issues:

- in which way such safety contracts should be specified,
- how they can be derived, and
- in which way they can be utilised for reuse of safety case artefacts.

First, we define safety contracts as a specific type of contracts that deal specifically with component behaviours that are considered safety relevant. Moreover, we characterise such contracts as either strong or weak to support specification of behaviours that reusable components exhibit in different systems in which such components could be used. More specifically, strong contracts capture behaviours that should hold in all systems in which the component can be used, while the weak contracts capture system-specific behaviours that are required to hold only in systems that satisfy both all the strong contracts and the corresponding weak assumptions.

Next, we investigate methods for deriving safety contracts from failure analyses. Just as hazard analysis is the basis for safety engineering at the system level, derivation of contracts and identification of the corresponding as-

sumptions plays a similar role at the component level [16]. As the basis for the safety contract derivation we use failure analyses recommended by the safety standards, e.g., Fault Tree Analysis (FTA) and Failure Mode, Effects and Criticality Analysis (FMECA). Moreover, we use results of Failure Propagation and Transformation Calculus (FPTC) analysis for contract derivation as it enables automation of failure analyses such as FTA and FMECA.

Finally, we provide methods for generative reuse of the safety case artefacts by utilising the safety contracts. More specifically, since safety contracts deal with some of the information used in the safety arguments (e.g., failure behaviour), we use the safety contracts to semi-automatically generate system-specific safety case argument-fragments. Moreover, we define a safety contracts development process to define the role of safety contracts in the system lifecycle. We align the proposed process with the ISO 26262 safety process as a way to fill the gap between reuse and integration of safety elements in the ISO 26262 safety standard. We use a real-world automotive product-line scenario for demonstration of the process. We utilise safety contracts during the development of a safety element that is developed out-of-context and reused together with its accompanying safety artefacts within two construction equipment products that belong to the same product-line.

In this thesis we mainly address the two reuse scenarios where reuse is performed either during the evolution of a single system or within a family of products. We do not fully tackle the cross-domain reuse scenario in the thesis as this scenario requires consideration of compliance of a system and its accompanying artefacts according to the different domain-specific safety standards. Although the approach proposed in this thesis is applicable for cross-domain reuse, it needs to be extended to consider multiple standard compliance to fully support cross-domain reuse scenario.

1.1 Outline

This thesis is organised in two parts. The first part summarises the research as follows: In Chapter 2 we describe our research methodology and the thesis research goals. We introduce some basic concepts used throughout the thesis in Chapter 3. In Chapter 4 we present the concrete thesis contributions in more detail. In Chapter 5 we present related work, and finally, we bring the conclusions and future work in Chapter 6.

The second part of the thesis consists of a collection of papers. We now present a brief overview of the included papers.

Paper A (Chapter 7). *Strong and Weak Contract Formalism for Thrid-Party Component Reuse*, Irfan Sljivo, Barbara Gallina, Jan Carlson, Hans Hansson.

Abstract. Our aim is to contribute to bridging the gap between the justified need from industry to reuse third-party components and skepticism of the safety community in integrating and reusing components developed without real knowledge of the system context. We have developed a notion of safety contract that will help to capture safety-related information for supporting the reuse of software components in and across safety-critical systems. In this paper we present our extension of the contract formalism for specifying strong and weak assumption/guarantee contracts for out-of-context reusable components. We elaborate on notion of satisfaction, including refinement, dominance and composition check. To show the usage and the expressiveness of our extended formalism, we specify strong and weak safety contracts related to a wheel braking system.

Status: Published in Proceedings of the 3rd International Workshop on Software Certification (WoSoCer), IEEE, November 2013

My contribution: I was the main contributor of the work under supervision of the coauthors. My contributions include extension of the contract formalism for specifying strong and weak assumption/guarantee contracts and the case study performed on an airplane wheel-braking system example.

Paper B (Chapter 8). *Generation of Safety Case Argument-Fragments from Safety Contracts*, Irfan Sljivo, Barbara Gallina, Jan Carlson, Hans Hansson.

Abstract. Composable safety certification envisions reuse of safety case argument-fragments together with safety-relevant components in order to reduce the cost and time needed to achieve certification. The argument-fragments could cover safety aspects relevant for different contexts in which the component can be used. Creating argument-fragments for the out-of-context components is time-consuming and currently no satisfying approach exists to facilitate their automatic generation. In this paper we propose an approach based on (semi-)automatic generation of argument-fragments from assumption/guarantee safety contracts. We use the contracts to capture the safety claims related to the component, including supporting evidence. We provide an overview of the argument-fragment architecture and rules for automatic generation, including

their application in an illustrative example. The proposed approach enables safety engineers to focus on increasing the confidence in the knowledge about the system, rather than documenting a safety case.

Status: Published in Proceedings of the 33rd International Conference on Computer Safety, Reliability, and Security (SafeComp), Springer-Verlag, September 2014

My contribution: I was the main contributor of the work under supervision of the coauthors. My contributions include extension of the component and safety contract meta-model, an architecture of the argument-fragment to be generated, rules for generation of the argument-fragments and an application of the proposed method on a fuel-level estimation system.

Paper C (Chapter 9). *A Method to Generate Reusable Safety Case Fragments from Compositional Safety Analysis*, Irfan Sljivo, Barbara Gallina, Jan Carlson, Hans Hansson, Stefano Puri.

Abstract. Safety-critical systems usually need to be accompanied by an explained and well-founded body of evidence to show that the system is acceptably safe. While reuse within such systems covers mainly code, reusing accompanying safety artefacts is limited due to a wide range of context dependencies that need to be satisfied for safety evidence to be valid in a different context. Currently the most commonly used approaches that facilitate reuse lack support for reuse of safety artefacts. To facilitate reuse of safety artefacts we provide a method to generate reusable safety case argument-fragments that include supporting evidence related to safety analysis. The generation is performed from safety contracts that capture safety-relevant behaviour of components within assumption/guarantee pairs backed up by the supporting evidence. We illustrate our approach on an airplane wheel braking system example.

Status: Published in Proceedings of the 14th International Conference on Software Reuse (ICSR 2015), Springer-Verlag, January 2015

My contribution: I was the main contributor of the work under supervision of B. Gallina, J. Carlson and H. Hansson. My contributions include derivation/translation of safety contracts from the results of the FPTC failure logic analysis, an extension of the method for generation of argument-fragments to provide better support for reuse of evidence and an application of the approach

on an airplane wheel-braking system example. The contributions of Stefano Puri include support for both modelling of the software architecture of the example and performing FPTC analysis in CHESS-toolset.

Paper D (Chapter 10). *Deriving Safety Contracts to Support Architecture Design of Safety Critical Systems*, Irfan Sljivo, 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.

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

My contribution: The first three authors were the main drivers of the work. My contributions include a method for derivation of safety contracts from Fault Tree Analysis and a method for completeness check of the contracts with respect to the fault trees. The contributions of Omar Jaradat include building of the safety case argument before and after introducing a change to the system, as well as capturing the connection between the derived safety contracts and goals in the safety case arguments to facilitate traceability mechanism between the system and its safety case.

Paper E (Chapter 11). *Using Safety Contracts to Guide the Integration of Reusable Safety Elements within ISO 26262*, Irfan Sljivo, Barbara Gallina, Jan Carlson, Hans Hansson.

Abstract. Safety-critical systems usually need to be compliant with a domain-specific safety standard, which in turn requires an explained and well-founded

body of evidence to show that the system is acceptably safe. To reduce the cost and time needed to achieve the standard compliance, reuse of safety elements is not sufficient without the reuse of the accompanying evidence. The difficulties with reuse of safety elements within safety-critical systems lie mainly in the nature of safety being a system property and the lack of support for systematic reuse of safety elements and their accompanying artefacts. While safety standards provide requirements and recommendations on what should be subject to reuse, guidelines on how to perform reuse are typically lacking.
We have developed a concept of strong and weak safety contracts that can be used to facilitate systematic reuse of safety elements and their accompanying artefacts. In this report we define a safety contracts development process and provide guidelines to bridge the gap between reuse and integration of reusable safety elements in the ISO 26262 safety standard. We use a real-world case for demonstration of the process, in which a safety element is developed out-of-context and reused together with its accompanying safety artefacts within two products of a construction equipment product-line.

Status: Technical report, ISSN 1404-3041 ISRN MDH-MRTC-300/2015-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, Sweden, March 2015

My contribution: I was the main contributor of the work under supervision of the coauthors. My contributions include the safety contracts development process and its application on a real-world case.

Chapter 2

Research Description

In this chapter we first describe the used research methodology and then present the research goal of the thesis, together with the corresponding research questions that have guided the work towards the specified goal.

2.1 Research Methodology

The goal of the research conducted in this thesis is to construct new methods, techniques and theoretical foundations based on the existing knowledge, in order to contribute to solving real-world problems. Such research, where solutions are designed and developed rather than discovered, is referred to as *constructive research* [17]. The nature of constructive research is in problem solving of real-world problems by providing solutions in form of new constructions that have both theoretical and practical contributions [18]. While the results of such research have both practical and theoretical relevance, the emphasis is placed on the theoretical relevance of the newly created construct.

The generic cycle of research within computing can be described in four high-level steps [19]. Figure 2.1 presents an overview of our adaptation of the four research steps. We first formulate the problem based on the current state-of-practice and state-of-the-art. After that we identify the gap in a current knowledge and propose a theoretical solution to bridge the gap. Next, we implement a practical solution based on the new theoretical constructs. Finally, we evaluate the implemented solution against the initially formulated problem.

Since the constructive research process [18] deals with both theoretical and

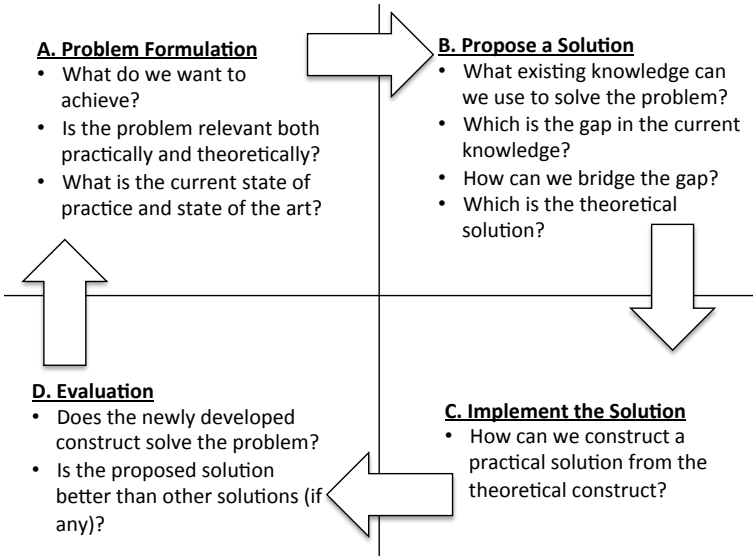

Figure 2.1: The cycle of the research process

practical problems, our research methodology consists of two nested instances of the generic research cycle, one cycle for the theoretical and one for the practical/engineering problem (Figure 2.2). The constructive research process starts with identifying a practically relevant real-world problem which at the same time has potential for theoretical contribution. The constructive work does not start until sufficient understanding of the research problem and the domain is obtained. The process of construction itself is where the main knowledge production usually happens [17]. In the second step the real-world problem is simplified and transferred into the research domain where we identify means for collecting data and establishing grounds for the constructive work. The second step starts the research/theoretical cycle where the research problem is refined into research questions. The next step includes the constructive work where a concrete research product is constructed as a solution to the research problem. In the subsequent steps of the research cycle, the proposed solution is implemented and evaluated against the research problem. Upon completion of the theoretical work, which is usually performed in several iterations for each of the research subgoals/questions, the practical cycle of the process continues to integrate the solutions to the research problem into a practical solution for

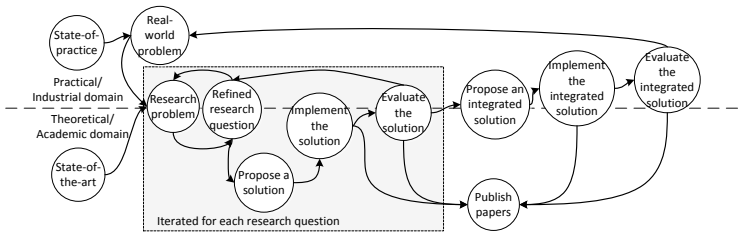

Figure 2.2: Overview of our research methodology

the real-world problem. In the next step the practical solution is implemented and finally, the practical solution is evaluated against the real-world problem.

Software engineering research often relies on case study methodology for both exploratory and evaluation purposes. Case study is an empirical method for investigating a contemporary phenomenon in its real-world context [20]. Exploratory case studies are usually conducted prior to the constructive work to gain deeper understanding of the research problem and the domain [17]. Upon development of new constructs, case studies can be used to evaluate a newly developed method or technique in its real-world setting. In some iterations of our research cycle we use case studies to evaluate the newly proposed methods.

2.2 Problem Statement and Research Goals

As mentioned in Section 1, safety-critical systems industries have problem with high production costs and the time needed to achieve safety certification for software-intensive systems. One way of addressing this issue is by enabling reuse of not only software components, but also the accompanying certification-relevant artefacts. The overall goal of the thesis is:

to facilitate systematic reuse of certification-relevant artefacts, i.e., safety case argument-fragments and the supporting evidence, related to the software components being reused.

The certification-relevant artefacts include safety argument-fragments and the supporting evidence used in those arguments to increase confidence in safety-relevant properties captured by the safety contracts, as mentioned in Section 1. Just as the components need to be designed for reuse, so do the

contracts as well. To achieve the overall goal we first establish a contract formalism to define how contracts should be specified to provide support for structured reuse of the captured safety-relevant properties.

Research question 1

How should safety contracts for software components be specified in order to facilitate systematic reuse of certification-relevant artefacts?

Safety-critical systems are characterised by a wide-range of properties on which the guaranteed safety behaviour of components depends, hence it is challenging to derive contracts with a sufficiently complete set of assumptions on the environment under which the contract guarantees hold. While many works deal with contract formalisms and how the contracts should look like, the issue of their derivation has not been sufficiently addressed. Just as failure analysis is the basis for safety engineering at system level, derivation of safety contracts and the corresponding contract assumption is the basis for safety engineering at component level. In our research we investigate how different failure analysis can be used to derive the safety contracts.

Research question 2

How can valid component safety contracts be derived from the results of different types of failure analyses to support systematic reuse of certification-relevant artefacts?

Safety is defined as an emergent property of a system, which means that we can only identify what is safety relevant for a particular system once we perform hazard analysis in the context of that system. Since reusable components are usually developed out-of-context of the system they are being reused in, it is difficult to reuse safety-relevant information with such components since we cannot know out-of-context what is safety-relevant for a particular system. In such cases, we can only speculate on what can be safety-relevant and include the information that can be potentially safety-relevant for the systems in which the component can be used. Since we use safety contracts to capture some of this information and specify the corresponding system dependencies in form of assumptions, we can use the contracts to identify the information relevant for a particular context. While the safety contracts capture some of this information (e.g., failure behaviour) in a more rigorous manner, the corresponding safety arguments present this information in a comprehensible way. The fact that both the safety argument about a component and the component safety

contracts deal with the same information makes the contracts an important aid in facilitating reuse of the argument-fragments and the supporting evidence. Moreover, as mentioned in Section 1, standards typically lack support for systematic reuse, which means that guidelines should be provided on how to use the safety contracts to perform reuse of the certification-relevant artefacts.

Research question 3

How can the component safety contracts be used to facilitate systematic reuse of the argument-fragments and supporting evidence?

Research questions on achieving better component interface descriptions in form of context-dependent specifications as well as reuse of safety case artefacts have been discussed in earlier research. To our knowledge, the originality of our research questions lays in combining the contractual specification of the component interfaces with reuse of safety case artefacts. More specifically, we focus on the context-dependent contractual specifications for reusable safety-relevant components and their derivation from failure analyses. We use such specifications to achieve generative reuse of safety case artefacts.

Chapter 3

Background

In this chapter we present an overview of safety-critical systems and their development. More specifically, we provide safety terminology, a classical system engineering safety process and a brief overview of safety standards. Moreover we provide essential information regarding representation of safety cases and Fault Tree Analysis, followed by a brief overview of reuse technologies used within safety-critical systems, in particular component-based software engineering, product line engineering and generative reuse. Finally, we provide an introduction to the notion of contracts and detail a contract theory that can be utilised for verification within safety-critical systems development, but also as means to support independent development and reuse within such systems.

3.1 Safety-Critical Systems

Safety is usually defined as *"freedom from unacceptable risk"* [21], where **risk** is a *"combination of the probability of occurrence of harm and the severity of that harm"* [21]. Since it is not practically feasible or possible to achieve absolutely safe or risk-free systems, acceptable levels of risk need to be established. Since risk itself is not accurately measurable, risk assessment is used to estimate levels of risk in order to "avoid paralysis resulting from waiting for definitive data, we assume we have greater knowledge than scientists actually possess and make decisions based on those assumptions" [22].

When dealing with risk, we distinguish between tolerable and residual risks. **Tolerable risk** is defined as *"risk which is accepted in a given con-*

text based on the current values of society" [21]. While the **residual risk** is defined as *"risk remaining after protective measures have been taken"* [21]. The protective measures are implemented by **safety functions** used to achieve or maintain a safe state in case a hazard occurs, i.e., to eliminate the hazards or to reduce the risk associated with the hazards to tolerable levels. A hazard is sometimes defined as a "potential source of harm" [21], but this definition is too generic, as almost any system state can be potential source of harm. Instead, a more concrete definition of a **hazard** is proposed, which defines a hazard as *"a system state or set of conditions that, together with a set of worst-case environmental conditions, will lead to an accident"* [23].

As can be noted, the definition of a hazard does not relate the hazard directly with a failure. In contrast to a hazard, a **failure** is defined as *"termination of the ability of a functional unit to provide a required function"* [21]. Hence there is a clear distinction between hazards and failures, since failures can occur without causing a hazard, and hazards can occur without any failures contributing to them. This is the crucial difference between safety and reliability [24], while safety deals with hazards, **reliability** deals with failures and is defined as *"continuity of correct service"* [25].

A system is composed of a set of interacting functional units used to implement system services. The **service** delivered by the system is *a set of external states of the system as perceived by its user* [25]. A service failure is caused by an **error**, which is *an external state of the service that deviates from the set of correct external states of the service* [25]. A cause of an error is called a **fault**. A fault is *an event that manifest itself in form of an error* [25]. Presence of an error in a system does not necessarily mean that the system will exhibit a failure. For example, a functional unit can be erroneous, but as long as that error is not part of the external state of the system, the system service will not exhibit a failure. In fact, there can be many errors in a system without it causing a failure, e.g., internal error checking can stop errors from propagating outside of the boundaries of the system. A failure occurs only if an effect of the error becomes observable outside of the boundaries of the system. An error that results in a failure can manifest itself in different ways. The way in which a functional unit could fail is called a **failure mode** [25].

3.1.1 Safety Standards

Safety-critical systems usually need to follow a set of specific rules and regulations that are mandated to assure that the residual risk has been reduced to acceptable levels. The rules and regulations, as well as the definition of the

acceptable levels of residual risk usually differ from one safety-critical domain to another. For example, the acceptable residual risks for a car and a plane crashing are not the same, hence the acceptable levels of residual risk differ between for instance avionic, automotive, railways and nuclear industries. This lead to differences in the rules and regulations to which systems within different industries need to be developed. These differences have resulted in a set of a domain-specific safety standard that provide recommendations and requirements on how safety-critical systems within specific domains should be developed.

Although the standards differ from each other, they share a common basis in the systems safety engineering process which is used to reduce or eliminate residual risk to acceptable levels by identifying and eliminating or controlling hazards. Such a process suggests that instead of safety being built into the system after system development, safety should rather be designed into the system from the beginning [26]. A **system** is defined as *"a combination, with defined boundaries, of elements that are used together in a defined operating environment to perform a given task or achieve a specific purpose. The elements may include personnel, procedures, materials, tools, equipment, facilities, services and/or software as appropriate."* [1]. Safety is considered a system property, as it can only be established in the context of a particular system. The safety process usually starts with understanding the system and its context.

The basis for safety engineering at the system level is hazard analysis and risk assessment [27], which usually follows after gaining sufficient knowledge of the system and its context. In this step, hazards are identified and associated with a level of risk. The acceptable levels of risk are usually established either based on a scale provided by the standard or by applying known risk acceptance principles such as As Low As Reasonable Practicable (ALARP). According to this principle, a risk is **ALARP** *"when it has been demonstrated that the cost of any further risk reduction, where the cost includes the loss of defence capability as well as financial or other resource costs, is grossly disproportionate to the benefit obtained from that risk reduction"* [1].

The next step focuses on developing means for reducing or eliminating the risks associated with the hazards by specifying safety requirements and developing the corresponding safety functions. A **safety requirement** is defined as *"a requirement that, once met, contributes to the safety of the system or the evidence of the safety of the system"* [1], while a **requirement** is defined as *an identifiable element of a function specification that can be validated and against which an implementation can be verified* [28].

A safety case is usually required in order to demonstrate that the system is

acceptably safe to operate in a given context. The safety case is composed of three main elements: safety requirements, safety argument and evidence [4]. The role of the safety argument is to clearly communicate the relationship between the requirements and the supporting evidence. The verification and validation activities are performed in order to collect the evidence that the system is acceptably safe in the given context. **Validation** is defined as *"the determination that the requirements for a product are sufficiently correct and complete"* [28], while **verification** is defined as *"the evaluation of an implementation of requirements to determine that they have been met"* [28].

Upon development of the system and successful assurance that it is acceptably safe, the safety process usually continues during the system operation with periodic audits and performance monitoring to ensure that the system is and remains acceptably safe.

Although conceptually there exists a common high-level systems safety engineering process, there are still different ways in which this process can be detailed and executed. The safety standards provide more detailed guidance for applying such a process in their particular domains. In the reminder of the section we briefly summarise some of the standards referred to in this thesis.

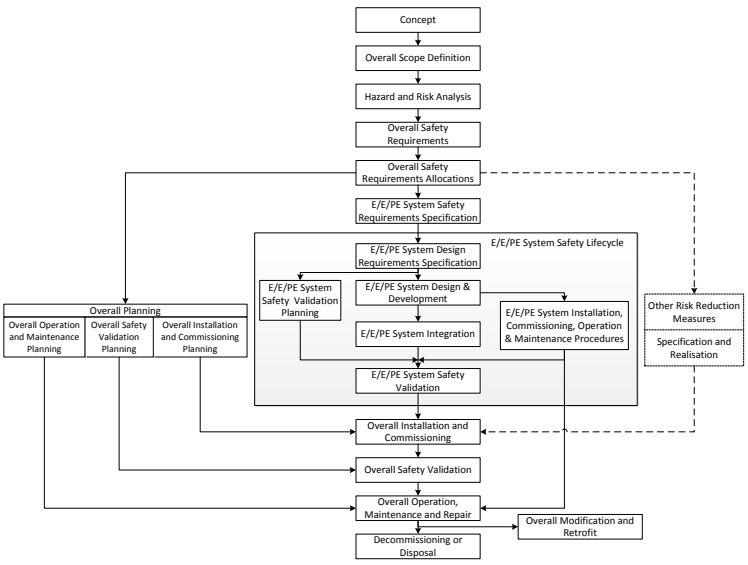

Figure 3.1: Overall Safety Lifecycle [29]

Generic Standard: IEC 61508

IEC 61508 is a generic standard for achieving safety of electrical/electronic/ programmable electronic systems [29]. IEC 61508 is published by the International Electrotechnical Commission (IEC) as a successor of its draft standard IEC 1508. The standard recognises that safety cannot be addressed retrospectively and that there is no absolute safety. Moreover the standard does not only address the technical aspects but it also includes activities such as planning and documentation as well as the assessment of all activities. This means that the standard does not only deal with system development but it encompasses the entire lifecycle of a system, from development, through maintenance to decommissioning. IEC 61508 is composed in such a way that it can either be applied directly or it can be further tailored for a specific domain. An overall safety lifecycle as indicated by this standard is shown in Figure 3.1. The presented lifecycle does not explicitly include verification activities, but they are required after each phase of the system development.

Railways Industry Standards: CENELEC EN 5012x

This group of standards for the railways industry represents the European Railways Standards required by law, and is composed of EN 50126 [30], EN 50128 [31] and EN 50129 [32]. These standards have been derived from the generic IEC 61508 standard. EN 50126 addresses the system issues and focuses on the specification and demonstration of reliability, availability, maintainability and safety (RAMS). EN 50128 provides guidelines and recommendations for which methods need to be used in order to provide software that meets the safety integrity demands placed upon it. EN 50129 addresses the approval process for individual systems and provides guidelines for demonstrating the safety of electronic systems and constructing the safety case for signalling railway application. EN 50129 explicitly requires a safety case to be provided and even defines its content.

Automotive Industry Standard: ISO 26262

Electronic and electrical systems (E/E) are more and more used to implement critical functions within road vehicles. In case of their malfunctioning behaviour many participants in traffic could be exposed to safety risk. The automotive industry safety standard ISO 26262 [33] has been developed as a guidance for how to provide assurance that any unreasonable residual risks

due to the malfunctioning of the E/E systems have been avoided. The standard is derived from the generic IEC 61508 standard and is composed of ten parts where the last part of the standard is dedicated to guidance on applying the standard. ISO 26262 provides requirements and recommendation on which activities should be performed as well as which work products should be provided for each of the activities covered by the standard. Moreover, it explicitly requires a safety case to be provided by progressively compiling it from the generated work products. The guidelines provided with the standard recommend provision of a safety argument within the safety case as a way to connect the generated work products with the safety claims about the system.

Civil Airspace Standards: DO 178(B/C), ARP 4754(A) and ARP 4761

The US Radio Technical Commission for Aeronautics (RTCA) and the European Organisation for Civil Aviation Equipment (EUROCAE) decided to develop a common set of guidelines for the development and documentation of software practices that would support the development of software-based airborne systems and equipment. The joint document was published as ED-12/DO-178 "Software Considerations in Airborne Systems and Equipment Certification" in 1982, followed by two revisions, revision A in 1985 and the second revision B in 1992 [34]. While RTCA publishes the document as DO-178(A/B), EUROCAE publishes the document as ED-12(A/B).

DO-178B addressed only the software practices and it requires an associated document for addressing the system-level activities. ED-79/ARP-4754 [28] "Certification Considerations for Highly-Integrated or Complex Aircraft System" was published in 1995 by SAE (Society of Automotive Engineers) and EUROCAE to address the total life cycle of systems that implement the aircraft-level functions. Since neither of the documents addressed the safety assessment methodologies, EUROCAE/SAE published a document ED-135/ARP-4761 [35] "Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment" specifically to address the methodologies for safety assessment processes.

The successor of DO 178B was made available in 2012. The document was published jointly by RTCA/EUROCAE as ED-12C/DO-178C [36] "Software Considerations in Airborne Systems and Equipment Certification" and is intended to replace the DO 178B standard. The new revision includes technology/method specific guidance with respect to model-based development and verification, object-oriented technologies, and formal methods. The revision A of the standard ARP 4754 related to the system-level activities was published

in 2010 with updates and extensions to the guidelines for the processes used to develop civil aircraft systems.

3.1.2 Safety Case Representation

As mentioned in Chapter 1, a safety case is composed of all the work products produced during the development of a safety-critical system, which includes a safety argument that connects the safety requirements and the evidence supporting and justifying those requirements. While the safety case represents the true reasoning as to why the system is acceptably safe, the safety argument is a representation of that reasoning aimed at communicating the actual reasoning as faithfully and clearly as possible [37]. Assurance case is a more generic term for cases where an argument is used to connect the requirements with the supporting evidence. An **assurance case** is 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."* [38]. A **claim** is defined as *"a proposition being asserted by the author or utterer that is a true or false statement"* [38], while an **argument** is defined as *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* [38].

The argument can be represented in different ways ranging from free text to more formal notations. The argument captures the rationale behind the produced artefacts and can take different form in different industries. It is referred to as a safety analyses report or safety case document/report where the outcomes of the safety process are summarised in natural language, while some use tabular structures to capture the rationale in structured form, others have started using graphical notations as they facilitate a more clear communication of the rationale and support hierarchical structuring of the rationale.

Free text has been the most typically used way of communicating the safety arguments within safety cases. While the free text might be more appropriate to use for simple cases, its limitations when used for more complex cases result in unclear and poorly structured natural language, which results in an ambiguous and unclear argument [4]. To overcome some of the limitations of creating safety arguments in free text, different approaches have been developed based on techniques such as tabular structures and graphical argumentation notations.

Tabular structures can be used to structure the safety arguments by representing the claims, argument and the evidence in different columns [1]. While the claim represents the overall objective of the argument (e.g., implementa-

tion is fault-free), the argument column represents a brief description of the type of the argument used (e.g., formal proof). The evidence column contains references to evidence or assumptions in form of assertions (e.g., proof tool is correct) that supports the stated argument description. The difficulty with tabular approaches is that they do not offer sufficient support for hierarchical structuring of the arguments, when used for complex arguments "clarity and the flow of the argument can be lost" [4].

To overcome the limitations of earlier approaches, graphical argumentation notations have been proposed to facilitate communicating a clear and structured argument. Currently there are two main approaches to representing the safety arguments graphically: Goal Structuring Notation (GSN) [4]; and Claims, Arguments and Evidence (CAE) [39]. Both approaches use similar elements for structuring the argument. In order to contribute to standardisation of the arguing techniques, a Structured Assurance Case Metamodel (SACM) [38] capturing the argument elements is introduced by the Object Management Group (OMG). The goal of the introduced metamodel is to allow for interchange of the structured arguments between different projects and tools by providing a standardised format for encoding safety arguments. We use GSN in our work to represent the safety arguments. In the reminder of the section we provide basic information about GSN.

Goal Structuring Notation

The Goal Structuring Notation (GSN) [40] is a graphical argumentation notation that can be used to record and present the main elements of any argument. The principal elements of GSN are shown in Figure 3.2. The main purpose of GSN is to show how **goals** (claims about the system), are broken down into **subgoals** and supported by **solutions** (the gathered evidence used to back up the claims). The rationale for decomposing the goals into subgoals is represented by **strategies**, while the clarification of the goals (their scope and domain) is done in the **context** elements. Justifications as to why a certain goal or strategy is considered appropriate or acceptable to use is done in the **justification** element. Validity of all the aspects that a certain goal or strategy depends on is not always argued over in the argument. Those aspects whose validity is not established in the argument but just assumed, are captured within the **assumption** element in form of assumed statements that should be checked outside of the argument. The argument elements can be connected with one of the two relationships: *supportedBy* and *inContextOf*. The **supportedBy** relationship is used to connect goals and strategies with other subgoals, strategies

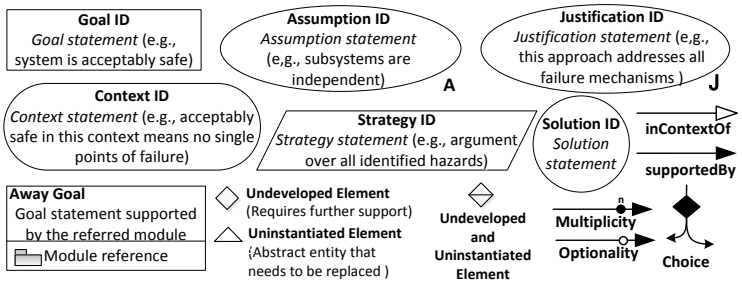

Figure 3.2: Overview of the GSN elements

and solutions, while the **inContextOf** relationship is used to connect the goals and strategies with supporting elements such as contexts, justifications and assumptions.

Since some arguments are developed using similar rationale in form of goals and strategies for decomposing those goals, reusable argument patterns are created by generalising the details of a specific argument. The main functionality of GSN has been extended to support representation of patterns of reusable reasoning [41]. To represent such argument patterns, GSN has been extended to support structural and entity abstraction [40]. The bottom row in Figure 3.2 presents some of the GSN extension elements. For structural abstraction the supportedBy relationship is extended by introducing multiplicity and optionality relationships. The **multiplicity** relationship indicates zero to many relationship between two elements, where *n* represents the cardinality of the connection. The **optionality** relationship indicates a zero or one cardinality connection between two elements.

To support entity abstraction, basic elements can be combined with uninstantiated and/or undeveloped elements. An **uninstantiated** element represents an abstract entity that is supposed to be replaced by a concrete element in the future. An **undeveloped** entity represents a concrete element that was not fully developed (e.g., not all subgoals are defined) and requires further development. The combination of the two indicates an entity that both needs to be replaced by a concrete element and needs to be further developed.

As one of the means to support modular extension to GSN, **away goals** are introduced to prevent repetition of parts of arguments across the different modules in which the argument can be partitioned [40]. Instead of further developing a certain goal, if that goal is already developed in another part of

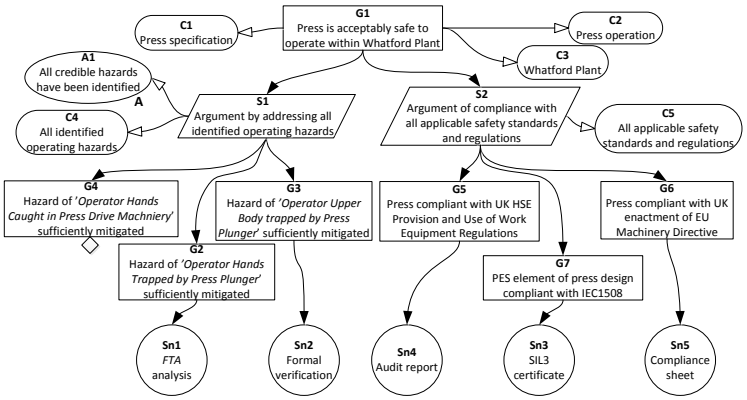

Figure 3.3: An argument example represented using GSN (adapted from [40])

the argument, an away goal can be used to point to the original element where the goal has been developed.

An example of the application of the core GSN is shown in Figure 3.3. The example presents a safety argument via GSN for a hypothetical factory that contains a metal press. The press has a single operator who inserts metal sheets, the machine presses the sheets to make car body parts and then the operator removes the parts from the press. The top-level goal *G1* argues that the press is acceptably safe to operate in the particular factory. The goal G1 is clarified with three context elements to explain different terms used in the goal statement. The goal is developed using two distinct strategies *S1* and *S2*. While the S1 strategy further decomposes the goal to argue over each identified hazard, the S2 strategy addresses compliance with different applicable standards. The assumption *A1* assumes that all credible hazards have been identified, which means that this statement will not be addressed in the argument. The strategies are further decomposed into corresponding subgoals that are then supported by different evidence in form of solutions. The goal *G4* is left undeveloped.

3.1.3 Fault Tree Analysis

Fault Tree Analysis (FTA) is one of the most commonly used techniques for preliminary system safety assessment where the design is examined to establish whether it achieves the allocated safety requirements. Moreover, FTA is

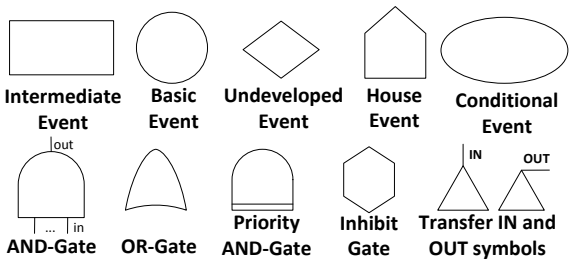

Figure 3.4: Main elements of the FTA graphical notation

often used to derive safety requirements [42]. FTA is a deductive failure analysis technique which focuses a single undesired event and methodologically determines the causes of that event [35]. It is a "top-down" approach used to construct a graphical representation of a model for an undesired event at the top of a hierarchical structure. The model represents a combination of basic events (modelled as leaves in the tree structure) that can lead to the top-level event. The analysis of the model can be done either qualitatively – by calculating cut sets, i.e., sets of basic events whose simultaneous occurrence will result in the top event, or qualitatively – where each basic event in the structure is assigned with a probability that is used to calculate the probability of the top-level event.

The main symbols of FTA are shown in Figure 3.4. The main symbols can be categorised as either logic gates or events, with an exception for the transfer symbols. The FTA terminology does not distinguish between faults, errors and failures, but all of those are represented as events in the tree. The logic gate symbols are used to connect different branches of the tree. Although there are several variations and extensions of the FTA graphical notation, we only describe the following fundamental symbols [35, 43]:

- **Intermediate Event**: An event that occurs because of one or more antecedent causes through logic gates
- **Basic Event**: A basic initiating event requiring no further development
- **Undeveloped Event**: An event which is not further developed either because it is of insufficient consequence or because information is unavailable
- **House Event**: An event which is external to the system under analysis, it will or will not happen (probability of occurrence 1 or 0)

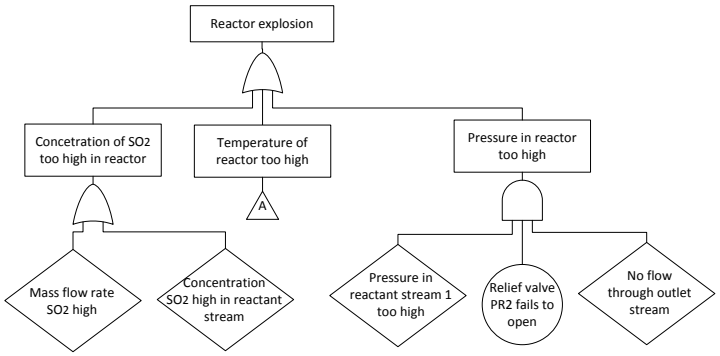

Figure 3.5: An example of a fault tree for a Chemical Processing System [44]

- **Conditional Event**: Specific conditions or restrictions that apply to any logic gate (used primarily with Priority AND-Gate and Inhibit Gate)
- **AND-Gate**: Output event occurs if all of the input events occur
- **OR-Gate**: Output event occurs if at least one of the input events occur
- **Priority AND-Gate**: Output event occurs if all of the input events occur in a specific sequence (the sequence is represented by a Conditioning Event drawn to the right of the gate)
- **Inhibit Gate**: Output event occurs if the (single) input event occurs in the presence of an enabling condition (the enabling condition is represented by a Conditioning Event drawn to the right of the gate)
- **Transfer In**: Indicates that the tree is developed further at the occurrence of the corresponding Transfer Out (e.g., on another page)
- **Transfer Out**: Indicates that this portion of the three must be attached at the corresponding Transfer In

An example of a fault tree is shown in Figure 3.5. The fault tree is used for analysing a Chemical Processing System [44] where the top-level event is *reactor explosion* that can be caused by either of the three related intermediate events. The *pressure in reactor too high* event can occur only if all three events related to it occur. While *Relief valve PR2 fails to open* is a basic event, the

other leaf intermediate events are not developed further in this example. The event *temperature of reactor too high* has a transfer-in symbol indicating that it is further developed elsewhere in the original document.

3.2 Reuse Technologies

Software reuse has been practiced since the first program was written. The paradigm for basing software development on reusable components dates back to the 1960s and Mcllroy's work [45]. **Software reuse** is defined as *"the use of existing software or software knowledge to construct new software"* [15]. Its main purpose is to improve both the quality of software and the productivity in creating the software. Artefacts subject to reuse can be either the software itself or knowledge related to the software. Such artefacts are referred to as **reusable assets**. **Reusability** is defined as *"a property of a software asset that indicates its probability of reuse"* [15].

As the programs got larger and more complex, means for systematic approaches to reuse had to be developed. The approaches to reuse are built on the following assumptions [46]:

- All experience can be reused;
- Reuse typically requires some modifications of objects being reused;
- Analysis is necessary to determine when, and if, reuse is appropriate.
- Reuse must be integrated into the specific software development;

The first assumption relates to the limitation of the traditional code-based software reuse and emphasises that all knowledge related to the code, including documents, processes, and all other software-related experiences should be subject to reuse together with the code. The second assumption relates to the fact that reuse "as is" is not likely. The reuse approaches need to consider that the reusable asset is likely to be modified once reused. The third assumption deals with identifying when is reuse appropriate and when does it pay off to reuse an asset. Some experiences indicate that in order to profit from reuse the software package needs to be reused at least three times [47]. The forth assumption implies that in order to achieve reuse, the reusable assets should be developed with reuse in mind, i.e., reuse methods and practices should be integrated in the software development process. We now briefly summarise some of the major reuse approaches evolved over years based on these assumptions.

3.2.1 Component-based Software Engineering

Building upon the reuse assumptions, the component-based development (CBD) approach emerged. The main idea of CBD is quick assembly of software systems from components already developed and prepared for integration. Despite many advantages [48], there are some limitations to the approach that have affected both customers and suppliers, who expected much more from CBD [49]. To address the limitations, a systematic approach to CBD that focuses on the component aspect of software development has been established as a new sub discipline of software engineering in form of Component-based Software Engineering (CBSE). The main goals of CBSE are to support the development of systems as composition of components, the development of reusable components, and to ease system maintenance and upgrades by simple component customisation and replacement [49]. CBSE inherently supports most of the reuse assumptions, although the assumption related to the analysis is not fully addressed within component-based technologies but is usually built into the development process model.

The central notion of CBSE is a component. Although many definitions of a component exist, the most widely used states that a **component** is *"a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third party"* [10]. Comparing the notion of a component to modules within modular approaches, where a **module** is considered as a set of classes or a package, a module *"does not come with persistent immutable resources, beyond what was hardwired in the code"* [50]. The components can be categorised as composite or atomic. An **atomic component** is *"a module plus a set of resources"* [50], while a **composite component**, or just a component, *is a set of simultaneously deployed atomic components* [50].

The most important element of a component is its interface. A component can implement a set of interfaces, which define the components access points. Each interface can consist of a set of operations used to provide services to other components. In contrast to interfaces, the implementation of the component must be encapsulated in the component and not reachable from the environment. For a component to be composable only based on its interface specification, such interface should be specified in a contractual manner. Besides the **provided interfaces** of a component, which *specify the operations the component implements*, to achieve the contractual nature of interfaces, **required interfaces** are defined to capture *the operations the component needs in order to function correctly* [50].

3.2.2 Product-line Engineering

Domain engineering also referred to as product line engineering or product family engineering, is an approach to systematic reuse built upon the reuse assumptions. It inherently supports all the reuse assumptions. It focuses on reuse of all domain knowledge, and is built to handle reuse of large adaptable components that are tailored for different products. Product lines do not appear accidentally, but they are planned as a consequence of a strategic decision of an organisation based on pay off analysis. Once an organisation decides to use the product line approach, their development process must integrate the product line engineering aspects. A **product-line** is defined as *"a top-down, planed, proactive approach to achieve reuse of software within a family (or population) of products"* [49]. This type of organised and planned approach to reuse is used within organisations that have **product families** that can be defined as *"a set of products with many commonalities and few differences"* [49].

A **software product line** is defined as *"a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way"* [51]. A **feature** is *"a logical unit of behaviour that is specified by a set of functional and quality requirements"* [52]. Features can be categorised as mandatory, optional or variable [53]. **Mandatory** features can be defined as *"core capabilities embodying the main domain characteristics at the problem level"*. **Optional** features indicate *"secondary properties of the domain"* [53] representing capabilities which are not necessary in some domains. **Variant** features represent *"alternative ways to configure a mandatory or an optional feature"* [53].

The basis of the software product line approach lies in the software architectural design in form of "a common architecture for a set of related products or systems developed by an organisation" [52]. Rather than satisfying requirements of a single system, a software product line architecture needs to satisfy requirements of the entire product family. The key process needed for the systematic design of the architecture is *Domain Engineering* [8]. Domain engineering starts with *domain analysis* which includes a systematic analysis of commonalities and variabilities across the product family. A **commonality** is a common product features across the product family, while a **variability** refers to behaviour of a reusable component that can be changed [8]. There are different **variability mechanisms** that can accommodate the change, e.g., inheritance, parametrisation of the component, and extension. The identification of the commonalities and variabilities leads to the definition of variation

points. A **variation point** identifies one or more locations at which a variation will occur in a product of the product family. The domain analysis is followed by an *application engineering* process for product derivation.

3.2.3 Generative Reuse

Another approach to systematic reuse that is tightly coupled with domain engineering is generative reuse. Unlike in CBSE, where reuse is based on generic components that are usually small in size, domain engineering approaches aim at achieving reuse of the domain specific components, which are generally larger in size. In general, reuse of the small generic components provides less reuse pay off than the reuse of the larger domain specific components [54]. Hence, generative reuse has the highest potential pay off, but it is the most difficult to achieve since the more components are domain specific the more they will suffer from reuse failure modes. Typical *reuse failure modes* [54] are: unacceptable performance of the reused component; lack of features or functionalities that would be difficult or impossible to add; and the reused component has incompatible interfaces or data structures with the target application.

While reuse can be achieved at different abstraction levels, the generative reuse is done at the specification level by means of application generators. We define **generative reuse** as *encoding of the domain knowledge into a domain specific application generator that generates new systems from specifications written for the new systems in a domain specific specification language* [15].

3.3 Contracts

Highlighting the contractual nature of interfaces in software engineering gained popularity with the Design by Contract technique developed for Object-Oriented Programming (OOP) by Meyer [55]. The notion of classes is fundamental in OOP where classes are defined as implementations of abstract data types. Design by Contracts presents classes as more than just a set of attributes and routines by including the semantic properties in form of assertions, to capture the true nature of the implementation. **Assertions** are boolean expressions that represent the semantic properties of classes and represent the basis for establishing the correctness and robustness of software [56], with **correctness** of software defined as *"the ability of software products to perform their exact tasks, as defined by their specification"* [56], and **robustness** defined as *"the ability of software systems to react appropriately to abnormal conditions"* [56].

Assertions are used in the fundamental notions of Design by Contracts: **preconditions** (*requirements under which routines are applicable*), **postconditions** (*properties guaranteed on routine exit*) and **class invariants** (*properties that characterise the class instance over their lifetime*). The precondition-postcondition pair for a routine describes a contract between a class and its clients (other classes that use the routine of the class). While the precondition binds the client (the caller of the routine), the postcondition binds the class (the supplier of a service through the routine).

The basis for Design by Contract are assertions that have been established within the works on program correctness by Floyd [57] and Hoare [58]. The Floyd-Hoare logic for proving partial correctness of sequential programs is represented by a formula in Hoare's logic $P\{S\}Q$, denoting that if assertion P is true before the initiation of the program S, the assertion Q will be true upon the termination of the program S. Moreover, the Design by Contracts notion of class invariants comes from Hoare's work on data types invariants [59], and their application to program design by Jones [60]. Furthermore, profound influence on the Design by Contract technique and the object oriented interface design was by Parnas and his work on information hiding [61, 62], and Dijkstra with his work that coined the phrase "separation of concerns" [63].

The pre- and postcondition contracts for sequential programs were extended to support concurrent programs by using rely/guarantee rules [64]. While the rely conditions make assumptions about any interference on the shared variables by the environment (during routine execution), the guarantee rules state the obligations of the routine regarding the shared variables.

To achieve the benefits of using the contracts in the context of CBSE, components should be enriched with the notion of contracts. This should allow the usage of third-party components in mission-critical systems. Beugnard et al. [65] propose a contract hierarchy that distinguishes between four levels of contracts. The first level are the basic contracts that represent the common interface notion, while the second level are the behaviour contracts in terms of pre- and postconditions in a sequential context. The third level are the synchronisation contracts that address the concurrent program execution aspects. The fourth level are the quality-of-service contracts cover the non-functional aspects of components (also commonly referred to as extra-functional properties or quality attributes). Non-functional properties are particularly important in embedded systems such as real-time systems and systems used within safety-critical domains, where properties such as timing, end-to-end deadlines, communication bandwidth or power consumption, play an important role.

The ultimate goal of attaching contracts to components is to support com-

position of systems through contract-based design which should contribute to system attributes such as correctness, robustness and reusability. While contracts for components can be established for different aspects (both functional and non-functional), combining contracts for different components and combining contracts for different aspects (viewpoints) attached to the same component requires different composition operators. This lead to development of a multiple viewpoint contract meta-theory [14] that provides mathematical foundations for contract-based model for embedded systems design. The theory is built upon the notion of the *heterogeneous rich components* (HRC) that encompass all the viewpoints necessary for electronic/electrical systems design. More specifically, it assumes a layered design space for electronic components (e.g., functional/software layer, ECU/hardware abstraction layer, and hardware-level layer) [14]. In the reminder of the section we provide the basic notions of the assumption/guarantee contract theory [11, 13, 14, 66].

3.3.1 Assumption/Guarantee Contract Theory

The component model of the assumption/guarantee contract theory is based on a set V of variables, where each **variable** represents the relevant information about a component (e.g., input and output ports) as a function of time. A component in such a model is described by an **assertion** P which can be modelled as a set of behaviours over the set V, more precisely the set of behaviours that satisfy the assertion. A **behaviour** (also referred to as a **trace** or a **run**) over the set V is a (finite or infinite) sequence of values assigned to each variable from the set V. Unlike assertions in form of preconditions and postconditions in program analysis which constrain the state space of a program at a particular point, assertions here are properties of entire behaviours.

A **contract** C is a pair C=(A, G) of assertions, called the assumptions (A), and the guarantees (G). All components E (representing the set of possible runs/traces of the component) that satisfy the assertion A, i.e., $E \subseteq A$, constitute the set of legal environments for the contract C. All components I that satisfy the assertion G provided that the assumptions A hold, i.e., $A \cap I \subseteq G$, constitute the set of all components implementing the contract C [14].

To simplify the definition of operators and relations, a contract C=(A, G) can be expressed in **canonical** (also referred to as normal or saturated) form by replacing the G with $G \cup \overline{A}$, (where $\overline{A}$ denotes the complement of A). The contract C and its version expressed in the canonical form are **equivalent**, i.e., they have the same implementations and environments [14].

For two contracts in canonical form, C=(A, G) and C'=(A', G'), the con-

tract C' **refines** the contract C, i.e., $C' \preceq C$, if and only if $A \subseteq A'$ and $G' \subseteq G$ [14]. For the contracts C and C' that are not in canonical form, a more thorough definition of refinement is established [11]. Both refinement definitions imply weakening of the assumptions and strengthening of the guarantees.

Conjunction of two contract C_1=(A_1, G_1) and C_2=(A_2, G_2) with the same sets of variables amounts to: $C_1 \cap C_2$=($A_1 \cup A_2, G_1 \cap G_2$). Which results in a contract that assumes all properties from both contracts and guarantees that satisfy both G_1 and G_2 assertions. This operation is used to compute the overall contract for a component from the component contracts associated with multiple viewpoints [14].

Composition of two contract C_1=(A_1, G_1) and C_2=(A_2, G_2) that are in canonical form and have the same sets of variables, results in a contract with assumptions A=$(A_1 \cap A_2) \cup \neg(G_1 \cap G_2)$ and guarantees G=$G_1 \cap G_2$ [14]. The resulting contract is in canonical form as well. While like for the conjunction operation, the resulting contract must satisfy the guarantees of both contracts, composing the assumptions is slightly different for the composition operation. Since composition is used on contracts that belong to different components, they are a part of each others legal environments and may already satisfy some assumptions of each other. More specifically, G_1 may satisfy some assumptions stated in A_2 and wise versa, hence guarantees of the two contracts under composition can contribute to relaxing the assumptions of the other contract under composition.

Chapter 4

Thesis Contributions

In this thesis we present a model for the formulation of the safety contracts and the related contract formalism, as well as means to use the safety contracts to facilitate systematic reuse of certification-relevant artefacts. We organise the contributions of the thesis in four parts. First, we adapt and extend the current contract formalisms [12, 14, 67] with the notions of strong and weak contracts to provide better support for reuse of certification-relevant data related to the contracts. Secondly, we present a method to derive the strong and weak safety contracts from the results of failure analyses. Then, we provide an approach to semi-automatically generate reusable safety case argument-fragments from safety contracts and in that way provide support for reuse of both the argument-fragments and supporting evidence. And finally, we propose a safety contracts development process to guide the reuse and integration of reusable safety elements by using safety contracts. In this section we briefly summarise these contributions.

4.1 Strong and Weak Contract Formalism

This contribution addresses the first research question: *"How should safety contracts for software components be specified in order to facilitate systematic reuse of certification-relevant artefacts?"*. The contract theory presented in Section 3.3 was used to build an assumption/guarantee contracts framework to include distinction between strong and weak assumptions and guarantees by extending the notion of a contract [68]. The extended contract is defined as

a tuple C=(A, B; G, H) consisting of the strong (A) and the weak (B) assumptions, and the strong (G) and the weak (H) guarantees. The idea with the extended contracts is that besides the strong assumptions that must be satisfied and the strong guarantees that are always offered, additional assumptions may be captured within the weak assumptions such that the weak guarantees are only offered when the weak guarantees are satisfied.

We present our extension of the contract framework to include specification of distinct strong and weak contracts and the related reasoning. Typically, behaviours of software components can be captured in assumption/guarantee contracts in such way that the component guarantees its behaviour if the stated assumptions on its environment are met. The distinction between strong and weak contracts allows for specification of properties that should hold in all systems of intended use of the component (strong contracts), and properties that can hold in a subset of the systems of intended usage (weak contracts). For a component to be used in a particular system, the strong assumptions of the component must be satisfied, and in return all strong guarantees are offered as a behaviour this component exhibits in all systems in which it can be used. The weak contracts describe behaviours that are achieved only in certain systems in which the component can be used. This distinction allows for identification of relevant behaviours for particular systems. For example, strong contracts can be used to prevent misuse of configuration parameters of the component by imposing parameters scope and guaranteeing behaviour achieved by setting the correct parameter value, while the weak contracts could be used to describe distinct component behaviours achieved by the different parameter values.

We denote strong contracts as $\langle A, G\rangle$ where A represents the strong assumptions and G the corresponding strong guarantees. The weak contracts are denoted as $\langle B, H\rangle$ where B represents the weak assumptions and the corresponding weak guarantees are denoted with H. A component C can be associated with a number of strong and weak contracts. For a component C to be used in a particular environment E all the strong assumptions of the related strong contracts of C should be satisfied by the environment E. We refer to such an environment as a correct environment of C. To ensure consistency between the strong and weak contracts, the weak contracts can only describe behaviours of the component C for a subset of the set of the correct environments. We say that a strong contract $\langle A, G\rangle$ holds if $A \wedge A \Rightarrow G$ is true, while the corresponding weak contract $\langle B, H\rangle$ holds if $A \wedge A \Rightarrow G \wedge B \wedge B \Rightarrow H$ is true.

The strong and weak contracts can be translated into traditional assumption/guarantee contracts. If all the traditional contracts of a component are

considered as strong contracts, then the weak contracts have to be specified as implications ($B \Rightarrow H$) within the guarantees of the traditional contract in order to avoid inconsistencies within the assumptions (e.g., two contracts that must be satisfied assuming different values for the same property). For example, the strong contracts of a component C are describing behaviours that hold for all products of a product-line, while the weak contracts are used to describe certain behaviours that hold only in a subset of the product variants of a product-line. Two of such weak contracts could be:

$$\langle \mathbf{B_1}\text{: productType=X; } \mathbf{H_1}\text{: accuracy is } a \rangle$$
$$\text{and}$$
$$\langle \mathbf{B_2}\text{: productType=Y; } \mathbf{H_2}\text{: accuracy is } b \rangle$$

The $\langle B_1, H_1 \rangle$ and $\langle B_2, H_2 \rangle$ contracts guarantee the different accuracies *a* and *b* of the component in two different product types *X* and *Y*. The translation of these contracts into a traditional contract would result in the following contract:

$$\langle \mathbf{A_t}\text{: -; } \mathbf{G_t}\text{: (productType=X } \textbf{implies} \text{ accuracy is } a) \text{ and}$$
$$(\text{productType=Y } \textbf{implies} \text{ accuracy is } b) \rangle$$

The contract checking in this case would not tell us which is the actual accuracy, only that the set of implications is guaranteed.

On the other hand, if all the traditional contracts of a component are considered as weak contracts, then in order to ensure that the strong assumptions are always satisfied they would have to be included in every contract of the component. For example, if we consider the following contract on the timing behaviour of a component:

$$\langle \mathbf{A_{t1}}\text{: Compiler=x and Platform=y and compilerConfiguration=z;}$$
$$\mathbf{G_{t1}}\text{: delay between } \textit{operation1} \text{ and } \textit{operation2} \text{ is less than XY} \rangle$$

we can't talk about the satisfaction of the timing contract unless the functional contracts of the component are satisfied, hence we would need to include the assumptions related to the functional contracts in each of the timing contracts. This would result in much more complex contracts, which would contradict the idea of dividing the contracts into viewpoints.

This contribution is discussed in detail in Paper A. Furthermore, the application of the strong and weak contracts on a real-world case is presented in Paper E.

4.2 Methods for Derivation of Safety Contracts from Failure Analyses

This contribution addresses the second research question: *"How can valid component safety contracts be derived from the results of different types of failure analyses to support systematic reuse of certification-relevant artefacts?"*. We present methods for deriving safety contracts from failure logic analyses such as Failure Propagation and Transformation Calculus (FPTC) and Fault Tree Analysis (FTA). Just as hazard analysis is the basis for safety engineering at the system level, derivation of contracts and identification of related assumptions plays a similar role at component level [16]. We use well-established failure logic analyses recommended by safety standards as the basis for contract derivation and assumptions identification. For example, FPTC [69] analysis allows for calculation of system level behaviour from the behaviour of the individual components established in isolation. The input/output behaviour of a component in isolation can be specified in FPTC rules. Once the component is instantiated in a context of a specific system, the system-level behaviour can be calculated. As this behaviour describes when it is safe to combine different components in the same system with respect to specific failure modes, it is worth capturing this behaviour in safety contracts. Such safety contracts describe two types of behaviours: (1) mitigation behaviour (e.g., if a component is designed to mitigate certain failures then the corresponding safety contracts should guarantee such mitigation behaviour), and (2) behaviours that lead to certain failures (e.g., if a component is not designed to mitigate certain failures then the corresponding safety contract capturing such behaviours establishes under which assumptions could such failure be avoided).

This contribution is discussed in detail in Papers C and D. While Paper C covers the derivation of safety contracts from FPTC analysis, Paper D focuses on the relationship between the safety contracts and FTA. Furthermore, the application of the contract derivation from the FPTC analysis on a real-world case is presented in Paper E.

4.3 A Method for Reuse of Safety Case Argument-fragments and Supporting Evidence

This contribution addresses the third research question: *"How can the component safety contracts be used to facilitate systematic reuse of the argument-*

fragments and supporting evidence?". We present a method for generation of reusable safety case argument-fragments from compositional safety analysis that allows for calculating the system-level behaviour from the behaviour of individual components (e.g., FPTC analysis). We first focus on the generation of context-specific safety case argument-fragments from safety contracts for components developed out-of-context. To generate an argument from safety contracts we define a set of specific aspects that should be covered in such an argument. For example, such argument needs to argue that the contracts are sufficiently complete, consistent and sufficient to address the corresponding safety requirements. Moreover, the argument should demonstrate that all the strong and relevant weak contract assumptions are satisfied. To support the generation of argument-fragments from safety contracts, we specify a component meta-model that focuses on an out-of-context setting and captures relationships between the safety contracts, safety requirements and the supporting evidence. Each safety requirement is addressed by at least one safety contract (either strong or weak), and each safety contract can be associated with the supporting evidence (including a supporting argument on e.g., tool qualification of the tool used to generate the evidence or personnel qualification of the engineers using the tool). To generate an argument-fragment, we first provide conceptual mapping of the argumentation notation elements with the component meta-model elements.

Once the contracts have been derived for a component developed out-of-context, the contracts are further supported by evidence such as failure analyses reports and additional V&V evidence. When the component is instantiated in a particular system, artefacts relevant for achieving safety goals of the particular system are identified through satisfied contracts and used in generation of the context-specific argument-fragment.

This contribution is discussed in detail in Papers B and C. Furthermore, the application of the contribution on a real-world case is presented in Paper E.

4.4 Safety Contracts Development Process

This contribution addresses the third research question: *"How can the component safety contracts be used to facilitate systematic reuse of the argument-fragments and supporting evidence?"*. We present a safety contracts development process and align it with the safety process recommended by the automotive ISO 26262 safety standard to provide guidelines on how to use the safety contracts to facilitate systematic reuse of Safety Elements out of Context. We

propose to divide the safety contracts development process in three phases: preliminary safety contracts, safety contracts production, and safety contracts utilisation and maintenance. During the first phase the strong and weak contracts are established, their assumptions are enriched with operational/environmental constraints, and each safety requirement allocated on a component is matched with at least one safety contract. Since not all information can be known at the stage of requirement specification that comes before the product development, some properties can only be established by speculation. For example, accuracy of the algorithm to be developed cannot be known before developing it, but only a speculative value can be established as a goal to be achieved. Hence this phase is called preliminary phase, since the contracts might need to be updated once the actual information is known.

The safety contracts production phase follows after the production of the component when the contracts from the preliminary safety contracts phase are actualised with implementation-specific properties. Moreover, in this phase the contracts are implemented with the component and supported by verification evidence gathered after the production of the component.

During the utilisation and maintenance phase, contracts should be checked to ensure that all the strong and the relevant weak contract assumptions are satisfied. In case the check fails, either the contracts should be reassessed, the corresponding component changed or the system adapted to achieve satisfaction of all the relevant contract assumptions. Once we have established that all the relevant contract assumptions have been satisfied, we utilise the contracts by applying the method for reuse of safety case argument-fragments and the supporting evidence.

This contribution and the application of the proposed process on a real-world case are presented in Paper E.

Chapter 5

Related Work

In this chapter, we relate the thesis contributions to similar relevant approaches. We provide a brief overview of, and comparison to, other contract-based approaches for safety-critical systems and approaches that aim at facilitating reuse of safety case artefacts.

5.1 Contract-based Approaches for Safety-Critical Systems

A range of formal contract-based approaches that focus on developing contract theories for assumption/guarantee contracts can be found in recent related work. The fundamental notions of such theories are presented in Section 3.3. Several approaches have been developed on top of the contract theories with focus on facilitating verification of the contracts for safety-critical systems. Damm et al. [67] demonstrate how contract-based component specification for different aspects of a component can be expressed using Requirement Specification Language (RSL). Moreover, the authors present how virtual integration testing of a composite component can be performed based on the contract-based specification of its sub-components. The approach proposed by Damm et al. categorises contract assumptions as either strong or weak to emphasise the methodological difference in the usage of different assumptions. In contrast, we focus on developing the notion of contracts for reusable components by categorising contracts as either strong or weak to clearly distinguish between assertions that must be satisfied whenever the component is used and

those that can be satisfied only in certain contexts.

In the approach by Gomez-Martinez et al. [70], the safety contracts are transformed in a series of steps into a formal model in terms of Generalised Stochastic Petri Nets to verify that the safety contacts have been satisfied. While in the work by Dragomir et al. [71], an extension to UML/SysML is proposed by providing language elements needed to model the contracts and their relations, with the purpose to facilitate compositional verification by using assume/guarantee contracts. These works build upon the traditional assumption/guarantee contracts and focus on compositional verification without considering out-of-context component development. In contrast, we aim at facilitating reuse of safety-relevant components and the accompanying artefacts by using strong and weak safety contracts.

Building upon the theoretical approaches, an approach by Söderberg and Johansson [72] to using safety contracts as safety requirements emerged. The assumptions and guarantees of the proposed safety contracts are composed of safety constraints such that each constraint is associated with a safety integrity level, just as a safety requirement. Another work by Westman et al. [73] focuses on structuring safety requirements by using assumption/guarantee contracts. This work relaxes the constraints of the underlying contract theory in order to capture the safety requirements allocated to a component in the guarantees of the corresponding component safety contracts. The assumptions of such a contract represent requirements on the environments of the component. In contrast to these works, we emphasise that there should be difference between the safety requirements and the content of the safety contracts if we want to use contracts to facilitate reuse. While a safety requirement describes behaviours a system requires from a component, the corresponding component contract guarantee that addresses the requirement should capture the actual behaviour of the component to which the safety requirement is allocated.

In another work by Battram et al. [74], a method for modular safety assurance based on assumption/guarantee contracts is presented. This work makes the distinction between interface and component contracts such that interface contracts are established between a component and its neighbouring components, while the component contracts are made between a component and its operating context. The work aims at easing the design of cyber-physical systems by using contracts to capture the requirements allocated to the component. The interface contracts can be useful for capturing the relationship between guarantees and assumptions of the neighbouring components in context of a particular system, but such contracts could not be captured out-of-context.

Adapting the classical contracts as defined by Meyer in the context of

Object-Oriented (OO) programming to fit component oriented programming requires lifting the contracts from the method level to the level of a component. Reussner and Schmidt [75] propose to align preconditions with the required interfaces and postconditions with provided interfaces. Moreover, since such contracts are not sufficient to represent quality attributes of components (such as reliability or performance), parametrised contracts are introduced as generalisations of the classical contracts by Reussner [76]. In our work we further extend the classical notion of contracts to provide fine-grained specification of safety-relevant properties for components developed out-of-context based on the trace-based contract theory.

Using the OO contracts for safety analysis can be done by defining a special type of safety contracts for OO systems to capture derived safety requirements (DSRs) by Hawkins [77]. Such contracts do not any more specify the expected behaviour as the classical OO contracts, but only the behaviour which is required to ensure that the corresponding object does not contribute to a particular hazardous software failure mode. Hawkins proposes to incorporate the behaviours specified by the DSRs into the design through the safety contracts to ensure that the software will not exhibit the identified unsafe behaviour once the design is implemented. In contrast, since we focus on components developed out-of-context we define safety contracts as those that capture behaviours deemed relevant from the perspective of hazard analysis. Moreover, to facilitate reuse we capture the actual behaviour of the components in the safety contracts, rather than the behaviour specified by the DSRs.

Except for partial support in work by Damm et al. [67] through introduction of strong and weak assumptions, these works do not provide support for capturing safety-relevant information for reusable components. Unlike in our work, none of these works actually focuses on how the contracts should be specified and used to support systematic reuse of software components together with the accompanying safety case artefacts. To the best of our knowledge the contribution of our work is in this respect novel and unique.

5.2 Safety Case Artefacts Reuse

There has been many works on modularising representation of safety cases in form of safety arguments and automating generation of the corresponding safety case artefacts in order to reduce the cost and time needed to compile a safety case. Fenn et al. [78] present an approach to incremental certification that uses “informal” contracts for generation of modular safety case argu-

ments. The approach uses Dependency-Guarantee Relationships (DGRs) that correspond to assumption/guarantee contracts. An argument for a module is derived by using all the DGRs of the module and their dependencies to other modules. In contrast, we base our work on a contract theory that does not limit the properties that can be captured within assumptions and guarantees to only two modules addressed by the DGRs.

One of the ways to reduce the cost and time needed to compile a safety case is by automatising generation of the safety case arguments. The works by Armengaud [79] and Basir et al. [80] focus on automating the compilation of the safety case arguments from pre-existing work products. Denney and Pai [81] focus on automating the assembly of safety cases based on the application of formal reasoning to software. The assembly combines manually created higher-level argument-fragments with an automatically generated lower-level argument-fragments derived from formal verification of the implementation against a mathematical specification. The work uses the AutCert tool for formal verification where the provided specification represent formalised software requirements. In contrast, we automate the generation of safety-case arguments from safety contracts that encode the domain knowledge, which in turn does not require re-generating argument after every change. Moreover, by capturing the assumptions that need to hold for information from a work-product to be guaranteed, we can highlight if the work-product needs to be revisited after a change to the system is introduced that violates the underlying assumption.

A work by Prokhorova [82] relies on formal modelling techniques supported by the Event-B formal framework. The work proposes a methodology for formalising the system safety requirements in Event-B and deriving a corresponding safety case argument from the Event-B specification. The work classifies safety requirements by the way they can be represented in Event-B and proposes a set of classification-based argument patterns to be used for generating specific arguments for each of the requirements classes. In contrast, we build upon contract-based specification that allows for capturing additional information besides the formalised requirements, which allows us to support generation of context-specific argument-fragments for reusable components.

Hawkins et al. [83] propose a model-based approach for standardising the representation of the assurance cases by generating it from automatically extracted information from the system design, analysis and development models. The proposed approach aims at ensuring the consistency in generation of the assurance case from the variety of sources from which the assurance case information needs to be extracted. In contrast, we use safety contracts and the related constructs to capture the assurance case information and its dependen-

cies to the artefacts from which this information is extracted, which provides the basis for reusing information gathered during the development of safety components ouf-of-context.

Habli [84] proposes a model-based assurance approach for facilitating reuse of safety assets within a product-line. Just as the product-line reference architecture is the base for deriving product architectures, the product-line safety case can play the same role for deriving an argument to why the particular product is acceptably safe to operate in the particular environment. The proposed approach for the product-line safety case development extends the argumentation notation to include product-line elements to handle variabilities within the argument. By capturing the variabilities and the underlying context assumptions, the approach can be used to reuse safety assets together with the used product-line assets. In contrast, instead of focusing on product-line engineering to achieve reuse of safety assets, we use contract-based specification to encode the safety reasoning and promote reuse of safety assets outside of a a family of products.

Certain safety-critical industries develop families of products that share certain product features, where the products must be developed according to different processes mandated by different safety standards, which in turn result in a family of safety cases to address each product of the family and the corresponding process. Gallina proposes to use a 3D product line for such scenarios to achieve reuse of all three aspects, the product, the process and the safety case [85]. The proposed approach combines a safety-critical product line – to promote reuse of the product features, a safety-oriented process line – which enables reuse of the process parts common between different processes mandated by different safety standards, and a safety case line – to promote reuse of the safety case artefacts generated from the corresponding process activities. In contrast, we do not focus on facilitating reuse only within a family of products, but aim at supporting reuse of out-of-context components that are not necessarily developed with a particular system in mind.

While these approaches offer a way to speed up the creation of a safety argument and reuse some of the safety case artefacts, they do not focus on reusable components developed out-of-context. Once such components are reused in a particular context, only artefacts relevant for that particular context should be reused and the corresponding argument generated. We use contracts to capture the context-specific dependencies of the component behaviours and in that way identify behaviours and the related artefacts that are relevant for the particular context in which the component is reused. To the best of our knowledge the contribution of our work is in this respect novel and unique.

Chapter 6

Conclusions and future work

In this chapter we first summarise and provide concluding remarks related to the research questions and thesis contributions, and then we present the future research directions.

6.1 Research Questions Revisited

The goal of our research is to facilitate reuse of safety-relevant software components and their accompanying safety case artefacts. As means for achieving our goal we focused on component assumption/guarantee contracts designed to support independent development of components and compositional verification. We specified three research questions (presented in detail in Section 2.2):

- **Research question 1**: "How should safety contracts for software components be specified in order to facilitate systematic reuse of certification-relevant artefacts?"
- **Research question 2**: "How can valid component safety contracts be derived from the results of different types of failure analyses to support systematic reuse of certification-relevant artefacts?"'
- **Research question 3**: "How can the component safety contracts be used to facilitate systematic reuse of the argument-fragments and supporting evidence?"

To offer answers to the specified research questions, we have presented a set of research contributions (detailed in Chapter 4):

- **Thesis contribution 1**: "Strong and Weak Contract Formalism"
- **Thesis contribution 2**: "Methods for Derivation of Safety Contracts from Failure Analyses"'
- **Thesis contribution 3**: "A Method for Reuse of Safety Case Argument-fragments and Supporting Evidence"
- **Thesis contribution 4**: "Safety Contracts Development Process"

Table 6.1 presents the mapping between the research questions and the possible answers we offered in form of the thesis contributions. In the reminder of this section we briefly summarise and provide concluding remarks for each of the research questions by reflecting on the corresponding thesis contributions.

Table 6.1: Mapping between the research questions and the thesis contributions

Questions	**Answers**
Research question 1	Thesis contribution 1
Research question 2	Thesis contribution 2
Research question 3	Thesis contributions 3 and 4

6.1.1 Research Question 1

Assumption/guarantee component contracts have been developed to support system design by focusing on capturing behaviour of the components in the particular system. The corresponding assumption/guarantee contract formalism was developed to support reuse through component selection, e.g., by capturing contracts for a component to be developed based on the behaviours of its environment, and then using such a contract to retrieve a component that satisfies that contract from a component library. The difficulty is that reusable components are usually developed in a configurable manner to support customising their behaviour to a specific system. The contracts need to be specified such that they provide better support for capturing such behaviours.

To support capturing the out-of-context behaviour exhibited by reusable components, we present a strong and weak contract formalism that allows for capturing component behaviours that are required to hold in all systems in which the component can be used (strong contracts), and system-specific behaviours exhibited only in a subset of the systems in which the component can

be used (weak contracts). By categorising contracts as strong or weak, we complement the original assumption/guarantee contract formalism by extending it to support development of reusable components developed out-of-context, where very little or no information is known about the contexts of the component. Strong and weak contracts for such reusable components represent a connection between the reusable components and the certification-relevant evidence produced to support the behaviour captured within the contracts. This connection provides the basis for supporting systematic reuse of certification-relevant artefacts.

6.1.2 Research Question 2

Safety contracts are a type of contracts that capture component behaviours deemed relevant from the perspective of hazard analysis. Just as the hazard analysis is basis for safety engineering on the system level, safety contract derivation and assumption identification plays similar role on the component level. To derive the safety contracts we show that failure logic analyses can be used to identify information that should be captured in the contracts. Moreover, we provide methods that can be used to derive contracts from failure logic analyses such as FTA and Failure Propagation and Transformation Calculus (FPTC). We use such well-established failure analyses to provide a clear methodology on how to derive safety contracts. The safety contracts provide a rich platform for specifying the failure behaviour, which allows for fully utilising and improving possibilities of automated generation of different failure analyses, such as FMEA and FTA.

6.1.3 Research Question 3

We propose a method for automatised argument-fragment generation from safety contracts and a safety contracts development process to provide guidance on how safety contracts can be used to facilitate systematic reuse of the argument-fragments and supporting evidence. In the reminder of this section we briefly summarise the two aspects of our answer to this research question.

Safety Case Argument-Fragment Generation

Since the safety contracts derived from failure logic analyses capture some of the crucial information for construction of a product-based safety argument, such contracts can be used for automating the generation of the core of the

product-based safety argument. Such argument should contain only information relevant for the specific system for which it is produced. We use safety contracts to semi-automatically generate such system-specific safety argument-fragments. To enable the generation of the safety argument-fragments from the safety contracts, we define a component meta-model that specifies a rich component composed of the safety contracts, the allocated safety requirements and the supporting evidence. Each allocated safety requirement should be supported by at least one safety contracts, which in turn should be supported by evidence. Then, we map the elements of the component meta-model with the elements of the argumentation notation, and provide transformation rules that generate an argument-fragment from the safety contracts and the related information. We specify the architecture of the argument based on safety contracts to show that the component is compatible with the particular system, that the relevant contracts have been satisfied and that they are consistent and sufficiently complete.

Safety Contracts Development Process

To use the safety contracts for reuse of safety case artefacts clear guidelines are needed to indicate how the contracts should be used within a typical safety process. We define a set of contract-specific activities in form of a safety contracts development process to provide such guidelines. Moreover, since the safety standards typically lack support for reuse, we align the proposed process with the automotive ISO 26262 safety standard to show how the safety contracts development process can be used to complement an existing safety process. We propose that the safety contracts development process is divided into three phases: preliminary phase where initial strong and weak contracts are captured and matched with safety requirements (done before the development of the product, hence such contracts may contain speculative/targeted behaviour); production phase where contracts are actualised with implementation specific-behaviours and supported by evidence (done during/after the production of the product); and utilisation and maintenance phase where the components are integrated with assistance of the contract verification, and then used for the generation of the corresponding safety argument-fragments (this phase is done in the context of a particular system). By providing such a process we were able to use it to demonstrate the usage of the safety contracts and the proposed methods on a real-world case example of a safety element out-of-context.

6.2 Future Research Directions

We have identified several research directions we would like to explore in more depth in the future:

- Strong and weak contract formalism optimisation
- Safety contracts language and patterns catalogue
- Safety case management
- Further safety case artefacts generation
- Further tool support

In the reminder of the section we briefly summarise each of the above future work directions.

6.2.1 Strong and weak contracts formalism optimisation

Currently we have only provided an extension of the existing contract formalism. This research direction needs to provide a complete formal foundations to our strong and weak contract formalism to optimise the performance to its dedicated use for handling behaviours exhibited by reusable components. Moreover, since we deem that capturing all possible assumptions required for a contract guarantee to hold is not always achievable, or not always desirable to achieve (e.g., it can be too expensive, time consuming, or not required), there is a need for establishing new techniques to analyse the safety contract completeness and consistency.

6.2.2 Safety contracts language and patterns catalogue

When deriving and specifying the safety contracts, there are patterns emerging for what the contracts contain, which assumptions are needed in certain cases and which guarantees should be combined with certain assumptions. This research direction focuses on either extending an existing (e.g., Othello Specification Language [86]) or providing a new contract pattern-based specification language to provide a catalogue of such assumption/guarantee contract patterns dedicated to capturing safety-relevant behaviour. Furthermore, methods for automatic identification of the assumptions (e.g., [87]) need to be extended to support identification of the distinct strong and weak assumptions.

6.2.3 Safety case management

Managing complexity of a safety case is becoming an issue. The safety arguments for even smaller systems can contain hundreds of goals, depending on the depth covered in the argument. While automating its generation reduces efforts and costs needed to achieve it, a big problem remains on assessing such an argument, since a safety assessor is required to examine it to determine whether the system is sufficiently safe or not. The aim of this research direction is to develop the contract formalism and the corresponding tools to a sufficient degree so that certain goals can be argued over implicitly, by providing sufficient confidence that they are achieved through the formalism.

Besides managing the complexity, the safety case change management is an important issue as well. The aim of this research direction is to develop techniques to perform change impact analysis on the safety case by using the safety contracts. Such techniques should allow for identifying which goals and their supporting evidence are affected by change and should be revisited.

6.2.4 Further safety case artefacts generation

In our research we have focused on using the safety contracts for generation of the safety-case argument-fragments. But the safety contracts can be used to generate other types of safety case artefacts, such as those possible to generate through FPTC analysis. This research direction focuses on developing techniques for generation of different failure analyses reports that can be used to present the information captured by the safety contracts in a more comprehensible manner for safety engineers to assess and further reason about the failure behaviour of the system. Moreover, this research direction includes work on developing additional techniques for analysing the safety contracts and utilising them to support optimisation of the architecture design of safety-critical systems.

6.2.5 Further tool support

Currently, we have only partial support for the strong and weak contract formalism within the CHESS-toolset[1]. This research direction focuses on further collaborations on extending the CHESS-toolset as well as some commercial tools to support the strong and weak contract formalism and the methods for generation and reuse of the safety case artefacts.

[1] http://www.chess-project.org/page/download

Bibliography

[1] UK Ministry of Defence (MoD). *Defence Standard 00-56 (Part 1)/4, Safety Management Requirements for Defence Systems*. Issue 4, UK Ministry of Defence, 2007.

[2] N. R. Storey. *Safety Critical Computer Systems*. Addison-Wesley, Boston, MA, USA, 1996.

[3] R. Bloomfield, J. Cazin, D. Craigen, N. Juristo, E. Kesseler, et al. Validation, Verification and Certification of Embedded Systems. Technical report, NATO, 2005.

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

[5] AC 20-148. *Reusable Software Components*. Federal Aviation Administration (FAA), 2004.

[6] ISO 26262-10. *Road vehicles — Functional safety — Part 10: Guideline on ISO 26262*. International Organization for Standardization, 2011.

[7] B. Meyer. The Next Software Breakthrough. *IEEE Computer*, 30(7):113–114, 1997.

[8] I. Jacobson, M. L. Griss, and P. Jonsson. *Software Reuse. Architecture, Process and Organization for Business Success*. Addison Wesley Longman, 1997.

[9] J. Varnell-Sarjeant, A. A. Andrews, and A. Stefik. Comparing Reuse Strategies: An Empirical Evaluation of Developer Views. In *8th International Workshop on Quality Oriented Reuse of Software*. IEEE Computer Society, 2014.

[10] C. A. Szyperski. *Component Software - Beyond Object-oriented Programming*. Addison-Wesley, 1998.

[11] S. 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 *15th international conference on Fundamental Approaches to Software Engineering*, FASE'12, pages 43–58, Berlin, Heidelberg, 2012. Springer.

[12] A. Cimatti and S. Tonetta. A Property-Based Proof System for Contract-Based Design. In *38th Euromicro Conference on Software Engineering and Advanced Applications*, pages 21–28. IEEE Computer Society, September 2012.

[13] I. Ben-Hafaiedh, S. Graf, and S. Quinton. Reasoning About Safety and Progress Using Contracts. In *12th International Conference on Formal Engineering Methods and Software Engineering*, ICFEM'10, pages 436–451, Berlin, Heidelberg, 2010. Springer.

[14] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. 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. Springer, 2007.

[15] W. B. Frakes and K. Kang. Software Reuse Research: Status and Future. *IEEE Transactions on Software Engineering*, 31(7):529–536, 2005.

[16] J. Rushby. Composing safe systems. In *8th International Symposium on Formal Aspects of Component Software*. Springer, September 2012.

[17] G. Dodig-Crnkovic. Constructive Research and Info-Computational Knowledge Generation. In *Model-Based Reasoning In Science And Technology – Abduction, Logic, and Computational Discovery (Studies in Computational Intelligence)*, pages 359–380. Springer, November 2010.

[18] K. Lukka. The Constructive Research Approach. In *Case Study Research in Logistics*, volume 1, pages 83–101. Turku School of Economics and Business Administration, 2003.

[19] H. J. Holz, A. Applin, B. Haberman, D. Joyce, H. Purchase, and C. Reed. Research Methods in Computing: What are they, and how should we teach them? *ACM Special Interest Group on Computer Science Education (SIGCSE) Bulletin*, 38(4):96–114, 2006.

[20] P. Runeson and M. Höst. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

[21] CENELEC. *IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems. Part 4: Definitions and abbreviations*. UK Ministry of Defence, 2007.

[22] W. D. Ruckelshaus. Risk, Science and Democracy. *Issues in Science and Technology*, 1(3):19–38, 1985.

[23] N. G. Leveson. *Engineering a Safer World: Systems Thinking Applied to Safety*. MIT Press, 2011.

[24] C. O. Miller. A Comparison of Military and Civil Aviation System Safety. In *Air Line Pilots Association Symposium*, December 1983.

[25] A. Avizienis, 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, 2004.

[26] N. G. Leveson. Software Safety: What, Why and How". *ACM Computing Surveys*, 18(2):125–163, June 1986.

[27] N. G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, Reading, Mass., 1995.

[28] Society of Automotive Engineers (SAE) and European Organisation for Civil Aviation Equipment (EUROCAE). *ED79/ARP-4754: Certification Considerations for Highly-integrated or Complex Aircraft Systems*. Society of Automotive Engineers, 1996.

[29] CENELEC. *IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems. Parts 1-7*. International Electrotechnical Comission, 2010.

[30] CENELEC. *EN 50126: Railway Applications The specification and Demonstration of Reliability, Availability, Maintainability and Safety (RAMS)*. European Committee for Electrotechnical Standardisation, Rue de Stassart 35, B - 1050 Brussels, 2007.

[31] CENELEC. *EN 50128: Railway Applications Communications, Signalling and Processing Systems Software for Railway Control and Protection Systems*. European Committee for Electrotechnical Standardisation, Rue de Stassart 35, B - 1050 Brussels, 2001.

[32] CENELEC. *EN 50129: Railway applications Communications, Signalling and Processing Systems Safety Related Electronic Systems for Signalling*. European Committee for Electrotechnical Standardisation, Rue de Stassart 35, B - 1050 Brussels, 2001.

[33] International Organization for Standardization (ISO). *ISO 26262: Road vehicles — Functional safety*. ISO, 2011.

[34] European Organisation for Civil Aviation Equipment (EUROCAE) and Radio Technical Commission for Aeronautics (RTCA). *ED-12/DO-178B: Software Considerations in Airborne Systems and Equipment Certification*. EUROCAE ED-12B and RTCA DO-178B, 1992.

[35] Society of Automotive Engineers (SAE) and European Organisation for Civil Aviation Equipment (EUROCAE). *ED-135/ARP-4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*. Society of Automotive Engineers, 1996.

[36] European Organisation for Civil Aviation Equipment (EUROCAE) and Radio Technical Commission for Aeronautics (RTCA). *ED-12C/DO-178C: Software Considerations in Airborne Systems and Equipment Certification*. EUROCAE ED-12C and RTCA DO-178C, 2011.

[37] I. Habli and T. Kelly. Safety Case Depictions vs. Safety Cases — Would the Real Safety Case Please Stand Up? In *2nd Institution of Engineering and Technology International Conference on System Safety*, pages 245–248. IET, 2007.

[38] Object Management Group (OMG). SACM: Structured Assurance Case Metamodel. Technical report, Version 1.0, OMG, 2013. http://www.omg.org/spec/SACM.

[39] ASCAD: The Adelard Safety Case Development Manual. Adelard, 1998. http://www.adelard.com/services/SafetyCaseStructuring/index.html.

[40] GSN Community Standard Version 1. Technical report, Origin Consulting (York) Limited, November 2011.

[41] T. Kelly and J. McDermid. Safety Case Construction and Reuse Using Patterns. In *16th International Conference on Computer Safety, Reliability, and Security*, pages 55–69. Springer, 1997.

[42] K. Hansen, A. Ravn, and V. Stavridou. From Safety Analysis to Software Requirements. *IEEE Transactions on Software Engineering*, 24(7):573–584, 1998.

[43] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. Fault Tree Handbook. Technical report, Springfield, January 1981.

[44] H. E. Lambert. *Fault Trees for Decision Making in Systems Analysis*. PhD thesis, Lawrence Livermore Laboratories, University of California, 1975.

[45] M. D. Mcllroy. Mass Produced Software Components. In *1st International Conference on Software Engineering*, pages 88–98. NATO Science Committee, 1968.

[46] V. R. Basili and H. D. Rombach. Support for Comprehensive Reuse. *IET Software Engineering Journal*, 6(5):303–316, September 1991.

[47] G. Caldiera and V. R. Basili. Identifying and Qualifying Reusable Components". *IEEE Computer*, 24(2):61–70, February 1991.

[48] A. W. Brown. *Large-scale, component-based development*, volume 1. Prentice Hall PTR Englewood Cliffs, 2000.

[49] I. Crnkovic and M. Larsson. *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA, 2002.

[50] C. A. Szyperski. Component Software and the Way Ahead. *Foundations of Component-Based Systems*, pages 1–20, 2000.

[51] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[52] J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*. Pearson Education, 2000.

[53] M. L. Griss, J. Favaro, and M. d'Alessandro. Integrating Feature Modeling with the RSEB. In *5th International Conference on Software Reuse*, pages 76–85. IEEE, 1998.

[54] T. J. Biggerstaff. A Perspective of Generative Reuse. *Annals of Software Engineering*, 5(1):169–226, 1998.

[55] B. Meyer. Applying 'Design by Contract'. *IEEE Computer*, 25(10):40–51, October 1992.

[56] B. Meyer. *Object-Oriented Software Construction, Second Edition*. The Object-Oriented Series. Prentice Hall, Englewood Cliffs (NJ), USA, 1997.

[57] R. W. Floyd. Assigning Meanings to Programs. In *American Mathematical Society Symposia in Applied Mathematics*, volume 19, pages 19–31, 1967.

[58] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *CACM: Communications of the ACM*, 12(10):576–580, October 1969.

[59] C.A.R. Hoare. Proof of Correctness of Data Representations. *Acta Informatica*, 1:271–281, 1972.

[60] C. B. Jones. *Software Development: A Rigorous Approach*. Prentice Hall International,, Hemel Hempstead (U.K.), 1980.

[61] D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *CACM: Communications of the ACM*, 5(12):1053–1058, December 1972.

[62] D. L. Parnas. A Technique for Software Module Specification with Examples. *CACM: Communications of the ACM*, 15(5):330–336, May 1972.

[63] E. W. Dijkstra. *A Discipline of Programming*, volume 1. Prentice Hall International, Englewood Cliffs, N.J., USA, 1976.

[64] C. B. Jones. Specification and Design of (Parallel) Programs. In *IFIP Congress*, pages 321–332, 1983.

[65] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *IEEE Computer*, 32(7):38–45, 1999.

[66] S. Graf and S. Quinton. Contracts for BIP: Hierarchical Interaction Models for Compositional Verification. In *27th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems*, FORTE '07, pages 1–18, Berlin, Heidelberg, 2007. Springer.

[67] W. Damm, H. Hungar, B. Josko, T. Peikenkamp, and I. 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. IEEE, 2011.

[68] A. Benveniste, J.-B. Raclet, B. Caillaud, D. Nickovic, R. Passerone, A. Sangiovanni-Vincentelli, T. Henzinger, and K. G. Larsen. Contracts for the Design of Embedded Systems, Part II: Theory. *Submitted for publication*, 2012.

[69] M. Wallace. Modular Architectural Representation and Analysis of Fault Propagation and Transformation. In *International Workshop on Formal Foundations of Embedded Software and Component-based Software Architectures*. Elsevier, 2005.

[70] E. Gómez-Martınez, R. J. Rodrıguez, L. E. Elorza, M. I. Rezabal, and C. B. Earle. Model-based Verification of Safety Contracts. In *1st International Workshop on Safety and Formal Methods*, volume 8938 of *Lecture Notes in Computer Science*, pages 101–115. Springer, 2014.

[71] I. Dragomir, I. Ober, and C. Percebois. Integrating Verifiable Assume/Guarantee Contracts in UML/SysML. In *6th International Workshop on Model Based Architecting and Construction of Embedded Systems*, volume 1084 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

[72] A. Söderberg and R. Johansson. Safety Contract Based Design of Software Components. In *3rd International Workshop on Software Certification, International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE Computer Society, November 2013.

[73] J. Westman, M. Nyberg, and M. Törngren. Structuring Safety Requirements in ISO 26262 Using Contract Theory. In *32nd International Conference on Computer Safety, Reliability, and Security*, volume 8153 of *Lecture Notes in Computer Science*, pages 166–177. Springer, September 2013.

[74] P. Battram, B. Kaiser, and R. Weber. A Modular Safety Assurance Method considering Multi-Aspect Contracts during Cyber Physical System Design. In *1st International Workshop on Requirements Engineering for Self-Adaptive and Cyber-Physical Systems*, 2015.

[75] R. H. Reussner and H. W. Schmidt. Using Parameterised Contracts to Predict Properties of Component Based Software Architectures. In *Workshop On Component-Based Software Engineering (in association with 9th IEEE Conference and Workshops on Engineering of Computer-Based Systems), Lund, Sweden, 2002*, 2002.

[76] R. H. Reussner. The Use of Parameterised Contracts for Architecting Systems with Software Components. In *6th International Workshop on Component-Oriented Programming (WCOP'01)*, June 2001.

[77] R. Hawkins. *Using Safety Contracts in the Development of Safety Critical Object-Oriented Systems*. PhD thesis, University of York, York, UK, 2006.

[78] 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.

[79] E. Armengaud. Automated Safety Case Compilation for Product-based Argumentation. In *Embedded Real Time Software and Systems*, February 2014.

[80] N. Basir, E. Denney, and B. Fischer. Building Heterogeneous Safety Cases for Automatically Generated Code. In *Infotech@ Aerospace Conference*. The American Institute of Aeronautics and Astronautics (AIAA), 2011.

[81] E. Denney and G. J. Pai. Automating the Assembly of Aviation Safety Cases. *IEEE Transactions on Reliability*, 63(4), 2014.

[82] Y. Prokhorova, L. Laibinis, and E. Troubitsyna. Facilitating Construction of Safety Cases from Formal Models in Event-B. *Information & Software Technology*, 60, 2015.

[83] R. Hawkins, I. Habli, D. Kolovos, R. Paige, and T. P. Kelly. Weaving an Assurance Case from Design: A Model-Based Approach. In *16th International Symposium on High Assurance Systems Engineering*, pages 110–117. IEEE, January 2015.

[84] I. Habli. *Model-Based Assurance of Safety-Critical Product Lines*. PhD thesis, University of York, York, UK, September 2009.

[85] B. Gallina. Towards Enabling Reuse in the Context of Safety-critical Product Lines. In *5th International Workshop on Product LinE Approaches in Software Engineering*. IEEE, May 2015.

[86] A. Cimatti, M. Dorigatti, and S. Tonetta. OCRA: A Tool for Checking the Refinement of Temporal Contracts. In *28th International Conference on Automated Software Engineering (ASE)*, pages 702–705. IEEE, November 2013.

[87] M. G. Bobaru, C. S. Pasareanu, and D. Giannakopoulou. Automated Assume-Guarantee Reasoning by Abstraction Refinement. In *20th International Conference, Computer Aided Verification (CAV)*, volume 5123 of *Lecture Notes in Computer Science*, pages 135–148. Springer, July 2008.

II

Included Papers

Chapter 7

Paper A: Strong and Weak Contract Formalism for Thrid-Party Component Reuse

Irfan Šljivo, Barbara Gallina, Jan Carlson, Hans Hansson.
In Proceedings of the 3rd International Workshop on Software Certification (WoSoCer), IEEE, November 2013

Abstract

Our aim is to contribute to bridging the gap between the justified need from industry to reuse third-party components and skepticism of the safety community in integrating and reusing components developed without real knowledge of the system context. We have developed a notion of safety contract that will help to capture safety-related information for supporting the reuse of software components in and across safety-critical systems.

In this paper we present our extension of the contract formalism for specifying strong and weak assumption/guarantee contracts for out-of-context reusable components. We elaborate on notion of satisfaction, including refinement, dominance and composition check. To show the usage and the expressiveness of our extended formalism, we specify strong and weak safety contracts related to a wheel braking system.

7.1 Introduction

More and more standards for certification of safety-critical systems are offering support for reuse of third-party components in order to reduce time-to-market and production costs. An example is represented by the introduction of the concept *Safety Element out of Context* (SEooC) within the automotive ISO26262 standard [1]. Although this opportunity to reduce time-to-market and production costs seems attractive, reuse of third-party components is challenged by various complications [2] and can easily incur additional costs. One of the major problems for the safety-related systems is that the context in which the reusable out-of-context component is going to be used is unknown. On the one hand, if we include too much information about the context in the reusable component than it will be more difficult to reuse it in a different context.

Component reuse within standards is present through the use of commercial off-the-shelf (COTS) and modified off-the-shelf (MOTS) items. The drawback of the off-the-shelf items is usually that they lack the development process evidence required for certification by the domain-specific safety standards [3]. The basic idea behind SEooC is to bridge that gap by allowing the developer to first assume the safety-related requirements applicable to a component, and then to develop it to satisfy those requirements. Contract-based approaches emerge as one of the means to capture safety requirements and enable reuse and composition within safety-critical systems [4, 5, 6, 7, 8, 9]. A contract is a set of assumptions and guarantees where guarantees are provided by the component if assumptions are met by the component's environment.

In our work, we are looking into means for capturing as much as possible of safety-related information for reuse, but still to keep the component more flexible, i.e., reusable. We provide a further developed formalism for safety contracts with strong and weak reasoning that enables capturing information that need to hold for all contexts, i.e., that are out-of-context, and information that are more context-specific.

To improve reuse possibilities of software components by using contracts, just as components need to be designed for reuse, so do contracts as well. For these purposes it is beneficial to provide more expressive means of capturing information for reuse. In our previous work [10], we have introduced fine-grained contract extension with strong and weak contracts reasoning. The strong contracts must always hold in order for components to be reused (in any context), while the weak contracts just offer additional information about a context in which the component can operate. For example, information such as timing are highly context-specific and should be specified separately from

the conditions that are needed for the component to operate.

In this paper we use a wheel braking system as an example of a safety-critical system to show how fine-grained contract reasoning can be used to capture timing and safety information. The system is originally used within ARP4761 airborne systems recommended practice [11] to demonstrate the safety process required for the airspace domain.

The main contribution of this work is extension and adaptation of contract semantics to handle strong and weak contracts. We associate each component with a set of strong and weak contracts and define conjunction of strong and weak contracts. The format of the contract in conjuncted form is based on our previous work [10], where a contract consists of strong assumptions, strong guarantees and multiple weak assumption/guarantee pairs. In this work, we define notions of satisfaction, refinement and dominance for contracts in the conjuncted format. Further more, we show the usage of the fine-grained contracts on the wheel braking system.

Comparing to related work, we are focusing more on capturing contracts for out-of-context components where very little, or no information at all is known about the context in which the component is supposed to operate. We are putting emphasis on the contents of out-of-context contracts and the separation of mandatory and alternative/optional properties.

The rest of the paper is organized as follows: In Section 7.2 we briefly present key notions we build upon and provide essential information on the wheel braking system. We extend and adapt the fine-grained contract formalism in Section 7.3. In Section 7.4 we use our extended formalism to specify contracts related to the wheel braking system. Related work is presented in Section 7.5 and conclusions and future work in Section 7.6.

7.2 Background

In this section we briefly provide some background information on off-the-shelf items for safety-critical systems, support for reuse from safety standards, and assumption/guarantee contracts. Finally, we also provide essential information related to the wheel braking system.

7.2.1 Off-The-Shelf Items

Off-the-shelf (OTS) solutions offer reduced time-to-market and increased affordability, and are expected to support services with multiple safety-criticality

levels [12]. There are many types of OTS items including commercial OTS, modified OTS, Software of Unknown Pedigree (SOUP) etc. While ones are developed according to standards - COTS and MOTS, the others are not - SOUP. On the other hand, some are to be used ”as is” without changes - COTS, and some can be modified and changed - MOTS.

The use of OTS items within safety-critical systems has been debated for years [3], since most of the safety-critical systems need to be certified by a domain-specific safety standard that requires some kind of evidence about the safety of the system, that usually doesn't come with OTS items. As all the other reusable components, OTS items as well suffer from the three basic issues in the creation and use of reusable components illustrated by the well-known ”3C's Model” [13]: Concept, Content and Context. The third issue is the most problematic for the safety community when it comes to reusable components. The problem of context is usually addressed by the concept of *separation of concerns*, where different aspects of a component are kept as independent as possible to maximize the reuse potential of the component. Due to the system-wide nature of the safety-related properties it is impossible to completely separate concerns in the context of safety-related systems.

7.2.2 Safety Standards and Reuse

Safety standard authorities have been making an effort to bridge the gap between the separation of concerns and the system-wide nature of safety properties. As mentioned in the introduction, an example is represented by the introduction of the concept *Safety Element out of Context* (SEooC) within the automotive ISO26262 standard. A SEooC is a safety-related element which is not developed for a specific item, but is developed based on ”assumptions on an intended functionality, use and context” [1].

Within avionics domain, regulated by DO178B(C) safety standard, the regulatory agency introduced the concept of *Reusable Software Component* (RSC) [14]. The concept allows developers of RSC to satisfy only a part of requirements mandated by the safety standard, while the integrator of the developed RSC is expected to complete the safety standard objectives.

Besides the above-mentioned problem with separation of concerns, another problem that occurs is the criticality of the components developed out-of-context. Safety Integrity Level (SIL) represents a measurement for quantifying risk reduction. Different safety standards have different SIL categorizations that range from events that have no risk involved to events that may result in harm to human life and can be classified as hazardous and catastrophic. The

components not only need to be developed according to a safety standards, but they usually must be developed at a specific SIL. Within the automotive industry standards, this problem is addressed by ASIL decomposition, where ASIL is in fact Automotive SIL. ASIL decomposition allows a developer to use a component with a lower SIL by attaching a safety function with the same lower SIL and showing that the two are independent. The avionics industry defines five SILs and refers to them as *Design Assurance Level* (DAL). DALs are categorized from catastrophic failure conditions denoted with DAL A to failure conditions that have no effect on safety denoted with DAL E.

7.2.3 Fine-grained Contracts

Traditional assumption/guarantee contract is a pair of assertions $C = \langle A, G \rangle$ where a component makes assumptions A on its environment and if those assumptions are met it offers guarantees G in return. Contract semantics are defined in terms of environments and implementations. It is said that an environment satisfies a contract $C = \langle A, G \rangle$ if it provides all of the contract assumptions A. An implementation satisfies a contract C if provided the assumptions A it satisfies the guarantees G.

As we mentioned in our previous work [10], moving properties captured in-context to out-of-context reusable component is a difficult work because many implicit and hidden assumptions need to be identified about the specific context, for the property to hold out-of-context. That is why we extended the traditional contract-based formalism to allow for distinguishing between properties that are context-specific and properties that must hold for all contexts by adapting strong and weak contract reasoning. The extended contract format consists of strong assumptions and guarantees and multiple weak assumption/guarantee pairs. While the strong assumptions and guarantees must be satisfied always in order for component to be used, the weak pairs offer additional information in some specific contexts where besides the strong assumptions, the weak assumptions are to be met as well.

7.2.4 Motivating Example

In this subsection we provide essential information related to the wheel braking system that we use to show the usage and expressiveness of our extended formalism. This information is based on [11] and is taken from two previous works [8] and [7].

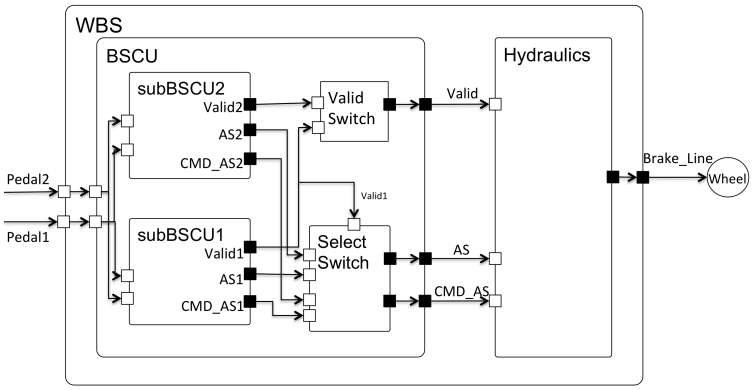

Figure 7.1: Wheel Braking System - High Level View

The example describes a Wheel Braking System (*WBS*) within an aircraft that takes two input brake pedal signals and outputs the brake signal that is applied on the wheel. The high level architecture is shown in Figure 7.1.

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 them to *Hydraulics* subsystem that executes the braking commands. If the *BSCU*, which makes the normal operation mode, fails then *Hydraulics* uses an alternate or emergency mode to perform the braking.

The *WBS* is designed so that it addresses requirement that loss of all wheel braking is less probable than 1.0E-7 per flight hour ("loss of all wheel braking" failure condition is classified as hazardous). In order to address the availability and integrity requirements imposed on *BSCU*, *BSCU* is designed with two redundant dual channel systems: *subBSCU1* and *subBSCU2*, shown in Figure 7.2. 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.

In this work we use contracts to capture safety and timing properties of the system. The timing requirement on the system is that its execution is no

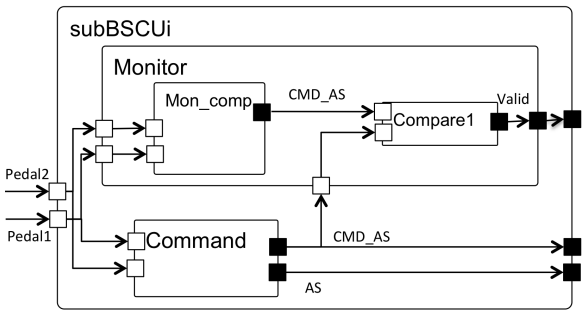

Figure 7.2: SubBSCUi

more than 10ms. We will detail more about what is needed to be assumed for this requirement to be guaranteed in the Section 7.4. The addressed safety requirement is that no single failure within the *BSCU* shall lead to "inadvertent braking due to *BSCU*".

7.3 Fine-grained contracts further development

In this section we extend the theoretical foundations of the fine-grained contracts presented in [10] and define contract relations and operations.

Contract-based approaches usually assume that a number of contracts in the form of assumption/guarantee pairs is attached to a component. The different contracts can be associated with different aspects or viewpoints of the system. We distinguish between properties that must hold in all contexts and properties that are context-specific by categorizing the contracts associated to components into strong and weak contracts. Strong contracts $\langle A, G \rangle$ are composed of strong assumptions (A) and strong guarantees (G), and weak contracts $\langle B, H \rangle$ of weak assumptions (B) and weak guarantees (H). While strong assumptions must hold in order for a component to be used in any context, weak assumptions and guarantees just provide additional information for particular contexts. The weak contracts ensure that in particular contexts satisfying the strong (A) and the weak assumptions (B), the component offers the weak guarantees (H).

For the purpose of defining operations and relations on the component contract we need to conjunct the weak and strong contracts to form a single component contract. For the conjuncted contract C we use the format presented

in [10]:

$$\langle A, G, \{\langle B_1, H_1\rangle, \ldots, \langle B_n, H_n\rangle\}\rangle$$

where A and G are above-mentioned strong assumptions and guarantees, and $\{\langle B_n, H_n\rangle\}$ represent a set of weak assumption/guarantee pairs i.e., weak contracts. Since all strong assumptions define a single set of environments ($\mathcal{E}_C$) in which the component can operate, we conjunct all strong assumptions into a single strong assumption (A) and all strong guarantees into a single strong guarantee (G). Each weak contract is valid in only a subset ($\mathcal{E}_n$) of that single set of environments defined by strong assumptions, i.e., $\mathcal{E}_n \subseteq \mathcal{E}_C$. Hence we don't conjunct weak contracts that are valid in different subsets but represent them in the contract as multiple weak assumption/guarantee pairs.

7.3.1 Contract relations and operations

For a contract C in the conjuncted form we say that an environment E is satisfying the contract if it satisfies the strong assumptions (A) i.e. if $E \in \mathcal{E}_C$. We refer to environments that satisfy the contract as correct environments. The set $\mathcal{E}_C$ is the set of all correct environments of contract C.

Any environment $E \in \mathcal{E}_C$ can satisfy some weak assumptions and be incompatible with others. The more weak assumptions there are satisfied by the environment E the more information about the component behaviour described by the weak contracts can be reused in this context.

The rich component concept we are assuming encompasses both implementation and contracts. We say that a component implementation I satisfies a contract $C = \langle A, G\{\langle B_n, H_n\rangle\}\rangle$ under two conditions: (1) implementation I satisfies the strong guarantees G in all correct environments of C, and (2) for all weak pairs within C, in all environments $E \in \mathcal{E}_C$ satisfying weak assumptions B_n, the implementation I satisfies corresponding weak guarantees H_n. An implementation I that satisfies the contract C is called a correct implementation of the contract C.

We assume a hierarchical component structure where a component can be primitive, i.e., atomic, or composite, i.e., consisting of subcomponents. By composing subcomponents we must ensure that resulting component implementation and contract holds. We check composition consistency of a composite component and its subcomponents with contracts in conjuncted form by (1) checking that the strong assumptions that are not satisfied by the interconnected subcomponents are assumed by the strong assumptions of the composite, and (2) checking that the composite contract follows from the subcomponent contracts.

For two traditional assumption/guarantee contracts C_1 and C_2 we get a composition contract by (1) composing assumptions that are not satisfied by the interconnected components, and (2) intersection of the guarantees. We compose two contracts in conjuncted form by (1) composition of the strong pairs, and (2) composition of the weak pairs such that there exists at least one environment satisfying the resulting weak pair within the set of all correct environments of the resulting contract, i.e., the intersection of the set of environments satisfying strong assumptions and the set of environments satisfying weak pair is not empty.

Relations of dominance and refinement are essential for checking composition and decomposition of contracts. We adapt the notion of dominance and refinement from [4] and [7] by including the weak and strong contract reasoning. Refinement coincides with weakening of assumptions and strengthening the guarantees within traditional assumption/guarantee contracts. While refinement of contracts in traditional form can be applied to strong and weak contracts individually, we say that refinement holds for two contracts C and C_1 in conjuncted form where C_1 refines C by (1) checking that the strong assumption/guarantee pair $\langle A_1, G_1 \rangle$ of C_1 refines strong assumption/guarantee pair $\langle A, G \rangle$ of C, and (2) that for each weak pair $\langle B, H \rangle$ within C such that B is related to C_1 there is at least one strong or weak contract within C_1 such that it refines or implies $\langle B, H \rangle$.

We say that composite component contract C dominates subcomponent contracts C_1 and C_2 if: (1) composition of any correct implementations of C_1 and C_2 forms a correct implementation of C, (2) for every subcomponent contract C' we say that correct implementations of other subcomponent contracts and a correct environment of the composite contract constitute a correct environment for the subcomponent contract C'. This means that for every weak assumption/guarantee pair $\langle B, H \rangle$within C such that B is related to a subcomponent C_n there is at least one strong or weak contract within C_n that implies or refines it. With our updated correct environment and implementation notions, conditions for checking dominance stay the same as in [4] and [7].

7.4 Case Study

In this section we show the usage and expressiveness of the extended contract formalism using the strong and weak contracts on the wheel braking system described in Section 7.2.4. We show that specifying contracts to provide better support for reuse requires additional constructs for contract specification

A: *Pedal1==Pedal2*
G: -

{⟨**B1:** Platform==x and Compiler==y;
H1: Delay between ($Change$(*Pedal1*,*Pedal2*), $Change$(*Brake_Line*)) $\leq$ 10ms⟩;}

Figure 7.3: WBS contract

and that the proposed contract formalism is more expressive for capturing this information.

In this example we use contracts to capture safety and timing analysis of WBS. We use contract language based on patern-based Requirement Specification Language as used in [8] and Othello System Specification from [7]. For specifying timing properties we use $Change(\{P\})$ for the event of change of the ports $\{P\}$, and $Delay\ between(p1, p2)$ to specify delay between the two changes $p1$ and $p2$.

As mentioned in our previous work [10], capturing timing information for reuse purposes requires additional constructs in identifying a set of assumption required for the reused timing information to be valid. In our work, we specify timing properties for reusable components within weak contracts i.e., weak assumption/guarantee pairs, because the timing information may be reused only if all of the assumptions related to timing are met, otherwise we can not reuse the behaviour of the component as described by the corresponding guarantees. Specifying timing information within weak contracts allows us to describe the timing behaviour of the component in different alternative contexts, e.g., timing properties of a component for different compilers or different compiler configurations. We present all contracts for this example in the conjuncted form.

7.4.1 Usage of the strong and weak contracts

The *WBS* component is composed of *BSCU* and *Hydraulics* subcomponents as shown in Figure 7.1, Section 7.2.4. The WBS component contract in Figure 7.3 describes a set of environments in which the system component can work by imposing the constraint that the two input pedals must be the same, otherwise all the mechanisms and redundancy within the WBS make no sense. As an additional information describing the component for certain correct environments, we make timing contract within weak assumption/guarantee pair by stating that for this particular platform, compiler and compiler configuration the component terminates within 10ms. This does not mean that the component

A: Pedal1==Pedal2
G: -

{⟨**B1:** (SubBSCU1.Valid or SubBSCU2.Valid);
H1: *BSCU.Valid* ⟩;
⟨**B2:** Platform==x and Compiler==y;
H2: Delay between ($Change$(*Pedal1*,*Pedal2*), $Change$(*CMD_AS*,*AS*)) $\leq$ 5ms⟩;}

Figure 7.4: BSCU contract

A: BSCU.Valid;
G:-

{⟨**B1:** Platform==x and Compiler==y
H1: Delay between ($Change$(*Valid*), $Change$(*Brake_Line*)) $\leq$ 5ms⟩;}

Figure 7.5: Hydraulics contract

shouldn't be used in an environment that doesn't satisfy those weak assumptions, but just that the timing behaviour of the component in that environment is known.

Figures 7.4 and 7.5 show *BSCU* and *Hydraulics* component contracts. While *BSCU* contract states that it requires the input pedals to be equal for the component to be able to operate, the *Hydraulics* contract requires only that the correct *Valid* signal from *BSCU* is received. We can see that composition of the subcomponents is correct since *WBS* takes *Pedal1==Pedal2* assumption from the *BSCU* subcomponent contract, since it cannot be satisfied by the interconnected components, while *BSCU.Valid* assumption from *Hydraulics* contract is satisfied by the interconnected *BSCU* component.

The timing contracts in Figures 7.3, 7.4 and 7.5 are defined for the same environment described by the assumed platform, compiler and compiler configuration. In order for the decomposition to be correct, the dominance should hold. The first condition for dominance specifies that correct implementations of *BSCU* and *Hydraulics* contracts form a correct implementation of *WBS* contract, which ensures that the timing contract of the composite is implied by the timing contracts of the subcomponents. Based on the refinement relation defined in Section 7.3.1, this way we imply that both related components *BSCU* and *Hydraulics* of *WBS* timing contract must either have a contract that refines or implies it. Refinement between the *WBS* contract and the subcomponent

A: *Pedal1==Pedal2*
G: -

{⟨**B1:** no fault in *Monitor*;
H1: *SubBSCUi.Valid* ⟩;
⟨**B2:** (*Monitor* developed to DAL A);
H2: *SubBSCUi.Valid* with high confidence⟩;
⟨**B3:** Platform==x and Compiler==y;
H3: Delay between ($Change$(*Pedal1*,*Pedal2*), $Change$(*Valid*,*CMD_AS*,*AS*)) $\leq$ 4ms⟩;}

Figure 7.6: SubBSCUi contract

A: *Pedal1==Pedal2*
G:*Monitor* developed according to DAL A;

{⟨**B1:** Platform==x and Compiler==y;
H1: Delay between ($Change$(*Pedal1*,*Pedal2*), $Change$(*Valid*)) $\leq$ 2ms⟩;}

Figure 7.7: SubBSCUi.Monitor contract

contracts holds since the strong pair is refined by the subcomponent contract strong pairs and the timing pair is refined and implied by the timing pairs of the subcomponents contracts.

In the previous works done on this system by [8] and [7], the safety requirement that "no single failure within *BSCU* shall cause inadvertent braking" is specified as "No_Double_Fault" variable meaning that always at least three out of four components within *BSCU* (two *Monitors* and two *Commands*) work correctly. This assumption is a direct representation of the requirement that no single failure shall cause BSCU to fail, by imposing too strict requirement

A: *Pedal1==Pedal2*
G: *Command* developed according to DAL B;

{⟨**B1:** Platform==x and Compiler==y;
H1: Delay between ($Change$(*Pedal1*,*Pedal2*), Change(*CMD_AS*,*AS*)) $\leq$ 1ms⟩;}

Figure 7.8: SubBSCUi.Command contract

A: -
G: always terminates;

{⟨**B1:** (Platform==x and Compiler==y) AND Valid1==TRUE;
H1: Delay between ($Change$(*CMD_AS1*,*AS1*), $Change$(*CMD_AS*,*AS*)) $\leq$ 0,25ms⟩;}
⟨**B2:** (Platform==x and Compiler==y) AND Valid1== FALSE;
H2: Delay between ($Change$(*CMD_AS1*,*AS1*), $Change$(*CMD_AS*,*AS*)) $\leq$ 1ms⟩;}

Figure 7.9: SubBSCUi.SelectSwitch contract

on the system. The BSCU can handle if more than one component of the four within *BSCU* fails, so for example if both *Command* components fail, the *Monitors* can still report the error and provide correct *Valid* signal. Another issue with the way safety contracts have been captured is separation of concerns. By using the "No_Double_Fault" variable on *WBS* level, authors are not respecting the encapsulation of *SubBSCUi* level by making assumptions about its internal structure on higher levels.

Our specification of the above-mentioned safety requirement can be seen through Figures 7.4, 7.6, 7.7 and7.8. We assume that no external fault is propagated through the pedal signals and that faults in this context refer to internal faults. In that case we can guarantee that the correct *Valid* signal will be provided by the *BSCU* if either of its subcomponents *SubBSCU1* or *SubBSCU2* provide the correct *Valid* signal, as assumed in *BSCU* contract in Figure 7.4. For the reasoning to hold, the subcomponents *SubBSCU1* and *SubBSCU2* guarantee the *Valid* signal only when the corresponding *Monitor* subcomponent is fault-free, Figures 7.6 and 7.7. To be able to guarantee *Valid* signal with a certain confidence, we capture the required DAL of *Monitor* within a weak pair. This way by respecting the separation of concerns and making assumptions only on assumed externally visible properties of interconnected components and subcomponents, we allow for extra flexibility of the system, hence better reuse possibilities.

On the primitive components level, that are not composed of subcomponents, we sometimes must make guarantees that are based on the external evidence i.e., guarantees that do not follow from assumptions. For example, for components *Monitor* and *Command* we make a statement/guarantee about the component that says that the component is developed according to a specific DAL. This information is essential when it comes to completing the safety contract structure. Some components can be considered reliable, such as *Se-*

lectSwitch, where we specified that it always terminates as its strong guarantee in Figure 7.9. The timing contract of *SelectSwitch* component specifies its behaviour for two alternative situations in which the component has different timing behaviour. It performs much faster when it just forwards the values by default from *SubBSCU1* (*Valid1* $== TRUE$), than when it must switch to the redundant component *SubBSCU2* (*Valid1* $== FALSE$).

7.4.2 Discussion on benefits of the extended formalism

Developing a new or moving an existing component to an out-of-context setting implies capturing increased number of assumptions and guarantees that are used to describe the behaviour of the component in different contexts, as we can see on the timing contracts examples from Section 7.4.1. Accordingly, the introduction of the additional constructs with the extended formalism offers us with possibility to specify conditions that are out-of-context i.e., that must be satisfied by any environment in order for component to operate or offer any kind of reuse in that environment (strong assumptions). Then, within all of those environments in which the component can operate and offer reuse, we have the possibility to specify conditions using weak contracts that describe behaviour of the component in some of the correct environments. For example, we use weak contracts i.e., weak assumption/guarantee pairs, to specify timing behaviour in different environments, or safety behaviour under different failure conditions, but all of these information can only be reused after the strong assumptions are met by the environment in which we use the component. As can be seen through contract examples in Section 7.4.1, this way of capturing context-specific information allows for extra flexibility of the system that enhances reuse possibilities.

7.5 Related Work

Contract-based design has been a research topic of many works in the recent years [6, 8, 4, 7, 9, 5]. These works are largely based on developing a theoretical foundation for contract-based framework and creating verification techniques for contract-based design. These works mainly focus on an Original Equipment Manufacturer (OEM)-supplier relationship. Through the examples provided in [8, 7, 5] and the formalisms [6, 4, 9, 5] we could notice the lack of focus on specifying contracts for out-of-context components that are planned to be instantiated or used in different contexts. When an OEM is developing a

component for a supplier, OEM usually has some requirements and demands about the component it needs to develop, which means that the context in which the component is supposed to operate is not completely unknown and many assumptions can be omitted since they are implied. When we want to actually move an in-context component to an out-of-context setting, or develop an out-of-context component, the number of properties that need to be captured increases and a more expressive way of capturing them is needed.

Comparing to the related work, in this paper we build on our previous work [10] and further extend the contract-based formalism [4] and [7] to provide more expressiveness for specifying contracts for out-of-context components. We additionally use an example that was used by [8] and [7] to show the usage of the strong and weak contracts for specifying out-of-context component contracts.

7.6 Conclusion and Future Work

We have presented our extended contract formalism for specifying strong and weak contracts to support reuse of safety-related information within safety-critical systems. We specify strong and weak contracts for components that are developed or moved to out-of-context setting, where very little or no information is known about the contexts of the component. We distinguish between properties that must hold for all contexts and properties that are more context-specific and are specified as additional or optional properties. The introduced additional constructs provide us with the possibility to capture context-specific information within contracts but still retain system flexibility that is needed to offer better reuse possibilities. Moreover, we define relations of satisfaction for implementations and environments in terms of strong and weak contracts, as well as relations of refinement and dominance between contracts. Finally, we use a wheel braking system as an example of a safety-critical system to demonstrate the usage and expressiveness of the extended formalism.

In our future work we plan to extend one of the existing contract languages such as Requirement Specification Language or Othello System Specification to support the presented extended formalism with strong and weak contracts. Further on, we see possibilities to establish a closer relation between the contracts in the presented form and safety argumentation used within the certification process of safety-critical systems.

Acknowledgements

This work is supported by the Swedish Foundation for Strategic Research (SSF) project SYNOPSIS and the EUs Artemis-funded SafeCer project.

Bibliography

[1] ISO 26262-10. *Road vehicles — Functional safety — Part 10: Guideline on ISO 26262*. International Organization for Standardization, 2011.

[2] O. Kath, R. Schreiner, and J. Favaro. Safety, Security, and Software Reuse: A Model-Based Approach. In *Proceedings of the 4th International Workshop on Software Reuse and Safety*, RESAFE '09, Washington, D.C., US, September 2009.

[3] F. Redmill. The COTS Debate in Perspective. In *Proceedings of the 20th International Conference on Computer Safety, Reliability and Security*, SAFECOMP '01, pages 119–129, London, UK, 2001. Springer-Verlag.

[4] S. 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.

[5] I. Ben-Hafaiedh, S. Graf, and S. Quinton. Reasoning About Safety and Progress Using Contracts. In *Proceedings of the 12th international conference on Formal engineering methods and software engineering*, ICFEM'10, pages 436–451, Berlin, Heidelberg, 2010. Springer-Verlag.

[6] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis. Multiple Viewpoint Contract-Based Specification and Design. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 5382 of *Lecture Notes in Computer Science*, pages 200–225. Springer, 2007.

[7] A. Cimatti and S. Tonetta. A Property-Based Proof System for Contract-Based Design. In Vittorio Cortellessa, Henry Muccini, and Onur

Demirörs, editors, *38th Euromicro Conference on Software Engineering and Advanced Applications*, pages 21–28. IEEE Computer Society, September 2012.

[8] W. Damm, H. Hungar, B. Josko, T. Peikenkamp, and I. Stierand. Using Contract-based Component Specifications for Virtual Integration Testing and Architecture Design. In *Design, Automation & Test in Europe Conference & Exhibition*, pages 1–6. IEEE, 2011.

[9] S. Graf and S. Quinton. Contracts for BIP: Hierarchical Interaction Models for Compositional Verification. In *Proceedings of the 27th IFIP WG 6.1 international conference on Formal Techniques for Networked and Distributed Systems*, FORTE '07, pages 1–18, Berlin, Heidelberg, 2007. Springer-Verlag.

[10] I. Sljivo, J. Carlson, B. Gallina, and H. Hansson. Fostering Reuse within Safety-critical Component-based Systems through Fine-grained Contracts. In *International Workshop on Critical Software Component Reusability and Certification across Domains*, June 2013.

[11] Society of Automotive Engineers (SAE) and European Organisation for Civil Aviation Equipment (EUROCAE). *ED-135/ARP-4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*. SAE, 1996.

[12] E. Kesseler. Assessing COTS Software in a Certifiable Safety-critical Domain. *Information Systems Journal*, 18(3):299–324, 2008.

[13] L. Latour, T. Wheeler, and W. B. Frakes. Descriptive and predictive aspects of the 3Cs model: SETA1 working group summary. In *Proceedings of the first international symposium on Environments and tools for Ada*, SETA1, pages 9–17, New York, NY, USA, 1991. ACM.

[14] AC 20-148. *Reusable Software Components*. FAA, 2004.

Chapter 8

Paper B: Generation of Safety Case Argument-Fragments from Safety Contracts

Irfan Šljivo, Barbara Gallina, Jan Carlson, Hans Hansson.
In Proceedings of the 33rd International Conference on Computer Safety, Reliability, and Security (SafeComp), Springer-Verlag, September 2014

Abstract

Composable safety certification envisions reuse of safety case argument fragments together with safety-relevant components in order to reduce the cost and time needed to achieve certification. The argument-fragments could cover safety aspects relevant for different contexts in which the component can be used. Creating argument-fragments for the out-of-context components is time-consuming and currently no satisfying approach exists to facilitate their automatic generation. In this paper we propose an approach based on (semi-)automatic generation of argument-fragments from assumption/guarantee safety contracts. We use the contracts to capture the safety claims related to the component, including supporting evidence. We provide an overview of the argument-fragment architecture and rules for automatic generation, including their application in an illustrative example. The proposed approach enables safety engineers to focus on increasing the confidence in the knowledge about the system, rather than documenting a safety case.

8.1 Introduction

The cost for achieving certification is estimated at 25-75% of the development costs [1]. As a part of certification, a safety case in form of a structured argument is often required to show that the system is acceptably safe to operate. To reduce cost and time-to-market, more and more safety standards are offering support for reuse within safety cases. Safety Element out of Context (SEooC) is an example of a concept for reuse proposed by the automotive ISO 26262 standard [2]. Building on such reusable elements, an approach to composable certification has been proposed [3]. The approach aims at achieving incremental certification by composing reusable argument-fragments related to safety elements, whose behaviour is specified through safety contracts. We define argument-fragments as parts of the system safety argument that argue about safety aspects relevant for the individual components.

In our previous work [4] we developed a safety contract formalism to facilitate reuse of components developed out-of-context. The safety contracts capture safety-relevant behaviours of the components in assumption/guarantee pairs. The semantics of such a pair is that if the assumption holds then the guarantee will also hold. The assumption/guarantee pairs are characterised as being either strong or weak. The strong contract assumptions are required to be satisfied in all contexts in which the component is used, hence the strong guarantees are offered in every context in which the component can be used. On the other hand, the weak contract guarantees are only offered in the contexts in which the component can be used and that satisfy the corresponding weak assumptions.

The strong and weak contracts allow us to distinguish between properties that hold for all contexts and those that are context-specific. Since every context has specific safety requirements, argument-fragments for out-of-context components may partially cover safety aspects relevant for several contexts. Creating argument-fragments for components developed out-of-context is a time-consuming activity. (Semi-)automatic generation of such argument-fragments from safety contracts would speed up the activity and allow for generation of context-specific argument-fragments. Moreover, the safety engineers would have the possibility to focus on increasing the confidence in the knowledge about the system, rather then on clerical tasks such as documenting a safety case [5].

Currently, no satisfying approach exists that facilitates generation of argument-fragments for out-of-context components. The main contribution of this paper is that we propose such an approach, capable to (semi)automatically gen-

erate argument-fragments from safety contracts and related safety requirements and evidence. As the basis for our approach we developed a meta-model that captures relationships between the safety contracts, safety requirements and evidence. To support the generation of argument-fragments from the safety contracts we provide conceptual mapping between the meta-model and argumentation notation elements. To perform the generation we provide the resulting argument-fragment architecture and a set of rules to generate the argument-fragments.

We demonstrate our approach on a Fuel Level Estimation System (FLES) and its variants that are used within Scania's trucks and busses. We focus on a single component of FLES that estimates the fuel level in the tank. This component represents a good candidate to be developed as SEooC as it is used with slight variations in many different variants. We use the safety contracts not only to capture the knowledge we have about the behaviour of the component, but also the evidence supporting the guaranteed behaviour. Moreover, by connecting in-context safety requirements with the weak safety contracts that address the requirements, we enable only those safety properties of the component relevant for the particular context to be used when developing the argument-fragment. This allows us to support more efficient creation of the argument-fragments as well as generation of context-specific arguments that contain information relevant for the context in which the component is used.

Compared to existing works, we focus on generation of argument-fragments for components developed and prepared for safety certification independently of the system in which they will be used. Approaches to generating safety case arguments [6, 7] usually extract the necessary information to build an argument from artefacts provided to satisfy some process, e.g., mandated by a safety standard. In our approach we utilise the safety contracts to capture the necessary information about a component from artefacts obtained out-of-context and show how argument-fragments can be generated for such components.

The structure of the paper is as follows: In Section 8.2 we present background information. In Section 8.3 we present the rationale behind our approach and how the generation of argument-fragments can be performed. In Section 8.4 we illustrate the approach for the Fuel Level Estimation System, and in Section 8.5 we provide a discussion of our approach. We present the related work in Section 8.6, and conclusions and future work in Section 8.7.

8.2 Background

In this section we introduce FLES that we use to illustrate our approach. We also provide some brief information on safety contracts based on our previous work; and Goal Structuring Notation, the argumentation notation we use for documenting safety case argument-fragments.

8.2.1 Illustrative Example: The Fuel Level Estimation System

In this subsection, based on [8], we provide brief but essential information related to FLES and the hazard analysis performed on it. We limit our attention to some bits of information that we use in illustrating the generation of argument-fragments.

FLES is based on a real estimation system used in Scania trucks with liquid fuel. The component-based architecture of FLES is shown in Fig. 8.1. The *Estimator* component estimates the volume of fuel in a vehicle's tank based on the sensor data obtained from the *Fuel Tank* and the *Engine Management System* (EMS). The received sensor values go through a series of transformations and filtering to handle any fluctuations in the sensed fuel level value. The estimated value is converted into percentage, passed to the *Presenter* and presented to the driver of the vehicle through the *Fuel Gauge* mounted on the dashboard. Due to dependencies of the transformations to the physical properties of sensors and its environment (e.g., size of the tank), these parameters are made configurable to make *Estimator* usable in different variants of the system.

The hazard analysis performed on the system reveals that if the fuel level displayed on the fuel gauge is higher than the actual fuel level in the tank then the vehicle could run out of fuel without the driver noticing, which would cause a sudden engine stop. If this happens while driving on e.g., a highway, the consequences could be catastrophic. Although there are other hazards in the system, this is the only hazard we use in illustrating our approach.

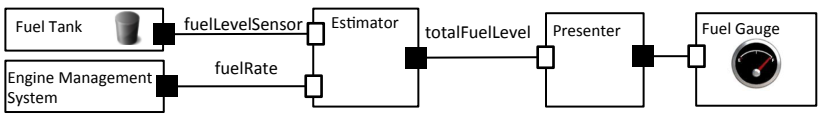

Figure 8.1: Fuel Level Estimation System

The safety analysis, as recommended by ISO 26262, starts by identifying at least one Safety Goal (SG) for each hazard, then for every safety goal, corresponding Functional Safety Requirements (FSRs) are derived and finally, Technical Safety Requirements (TSRs) are derived from the FSRs. We consider the following SGl and derived FSR:

- *SG1*: FLES shall not show higher fuel level on the fuel gauge than the actual fuel in the vehicle's tank;
- *FSR1*: Estimator shall not provide value of the estimated fuel level that deviates more than -5% from the actual fuel-level in the tank.

Additionally, the engine status signal provided by EMS should not be older than 0.3 seconds. An older value could result in a too high deviation from the actual fuel consumption that may cause deviation in the estimated fuel level value.

8.2.2 Strong and Weak Contracts

Our extension of the traditional contract-based formalism with strong and weak contracts allows for distinguishing between properties that are context-specific and properties that must hold for all contexts [9].

A traditional assumption/guarantee contract $C = \langle A, G \rangle$ is composed of assumptions A and guarantees G, where a component offers the guarantees G if its assumptions A on its environment are satisfied [10]. As an illustrative and simplified example based on the system we presented in Section 8.2.1, we specify a contract for Estimator with assumptions that if both the fuel level and fuel rate are provided with sufficient accuracy, Estimator guarantees that the total estimated fuel level it provides will be with certain accuracy.

Strong contracts $\langle A, G \rangle$ are composed of strong assumptions (A) and strong guarantees (G), and weak contracts $\langle B, H \rangle$ of weak assumptions (B) and weak guarantees (H) [4]. While strong assumptions must hold in order for a component to be used in any context, weak assumptions and guarantees just provide additional information for particular contexts. We say that a component, described by a set of safety contracts, is compatible with a certain context if all of its strong assumptions are satisfied by the environment. The weak contracts ensure that in all compatible contexts where the weak assumptions (B) are satisfied, the component offers the weak guarantees (H). For example, strong contracts could assume input type, range, or minimum amount of stack required and guarantee similar properties. On the other hand, weak contracts assume configurable parameters such as tank or sensor parameters in FLES and

guarantee different behaviour of the component dependant on those parameters such as different accuracy of the output or specific timing behaviour.

8.2.3 Goal Structuring Notation

In this paper, we use Goal Structuring Notation (GSN) [11] for expressing safety case argument-fragments. GSN is a graphical argumentation notation that can be used to specify elements of any argument. Some of the basic elements of GSN are illustrated in Fig. 8.2 and their semantics is given in the following list:

- *Goal*: a claim or a sub-claim that should be supported by the underlying argument. It can be broken down to several sub-goals (sub-claims).
- *Strategy*: describes a method used to develop a goal into additional sub-goals.
- *Context*: represents the domain/scope of the element it is connected to.
- *Solution*: describes the evidence that the connected goal has been achieved.
- *Undeveloped element*: states that the element to which the symbol is attached requires further development.
- *InContextOf*: used to connect context with goals.
- *SupportedBy*: used to show relationship of inference between goals in the argument, or to show that certain evidence is supporting a goal.
- *Away goal*: used to specify a module in which the goal is further developed.

For the sake of clarity it must be noted that the context element can be used to simply enrich or clarify the statements of the elements it is connected to. Besides the basic symbols, we additionally use a notational extension that supports abstract argument patterns [11]. More specifically, to denote a variable we use the curly brackets within statements; to denote generalised n-ary relationships between GSN elements we use the *supportedBy* relationship with a solid circle; to denote a choice, either 1-of-n or m-of-n selection, we use a solid diamond, which can be paired, using a simple connector line, with an *Obligation* element represented by an octagon symbol, stating condition for the choice selection.

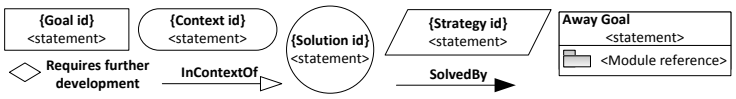

Figure 8.2: Basic elements of the Goal Structuring Notation

8.3 Composable Arguments Generation

The aim of this section is twofold: (1) to explain the rationale underlying our approach to (semi)automatic generation of argument-fragments, and (2) to explain how the generation can be performed. The latter is done by

- providing a component meta-model, developed to capture the relationships between the safety contracts, safety requirements and evidence in an out-of-context setting, and being sufficient to provide us with the information required for argument-fragment generation,
- presenting a conceptual mapping of the meta-model elements to a subset of the basic GSN elements to provide better understanding of the transition from the meta-model to the argument-fragment,
- presenting an overview of the argument-fragment architecture, and by
- providing a set of rules for the argumentation-fragment generation.

8.3.1 Rationale of the approach

In our work we focus on safety-relevant components developed and prepared for safety certification independently of the system in which they will be used. To develop such components, the engineer must assume some safety requirements that might be required when the component is used in a context. To prepare components for certification, safety engineers need to capture safety-relevant properties of the component that show how the safety requirements allocated to the component are met. To do that, we use our notion of strong and weak contracts.

It is worth to point out that the safety requirements and the safety contracts we use are closely related, but not the same. The safety contracts contain information about the actual behaviour of the component. On the other hand, the safety requirements contain information about what a particular context/system

requires from the component. While the safety requirements vary between contexts, the safety contracts should be correct regardless of the context. This is important to enable reuse of out-of-context components. As an illustration, consider FLES example requirement "Estimator shall send a valid value in totalFuelLevel within 2 seconds from when the Electronic Control Unit starts". This is a requirement on Estimator in this particular context and should not be specified within the Estimator's safety contract in that form. In the safety contract we should rather specify the actual time Estimator needs to send the totalFuelLevel. This makes the contracts independent of the context in which out-of-context component can be used, which allows us to use the knowledge captured within the contracts for all contexts in which the contracts are satisfied. The strong contracts denote properties that must be argued about in argument-fragments for every context, while the weak contracts will be argued about only if associated with a safety requirement within a particular context.

In order to guarantee the actual behaviour of the component, as specified in the safety contracts, we need to provide evidence about confidence in the contract. We categorise the evidence that supports the confidence in the contracts in terms of completeness, correctness and consistency, as follows: (1) completeness refers to whether contracts have captured all the needed properties of the component and the environment, (2) correctness refers to whether the contracts are correct with respect to associated requirements and (3) consistency refers to whether the contracts are not contradicting each other.

When using an out-of-context component in a particular context, a set of actual safety requirements (e.g., FSR or TSR) is allocated to the component. One of the roles of an argument-fragment is to show that these requirements are met. As safety contracts can be used to address different types of requirements, we are developing our approach without focusing on a particular class of requirements.

The (semi)automatic generation of argument-fragments from the safety contracts enables us to reduce the effort safety engineers need to dedicate for creating a set of argument-fragments. These fragments could be created for several contexts in which the component could be used. By speeding up both the integrator's and the developer's activities related to documenting a safety case, we enable them to focus on activities related to their knowledge about the system, by capturing this knowledge in the safety contracts.

8.3.2 Component meta-model

Our component meta-model in Fig. 8.3 is presented as an UML class diagram. This diagram captures the relationships between the assumed requirements, safety contracts and evidence, as described in Section 8.3.1. Our meta-model is based on the SafeCer component meta-model [12], which we have adapted, focusing only on its out-of-context part. Instead of associating argument-fragments (that may contain information not relevant for a specific context) with a component, we associate evidence and safety requirements directly with contracts to facilitate generation of context-specific argument-fragments.

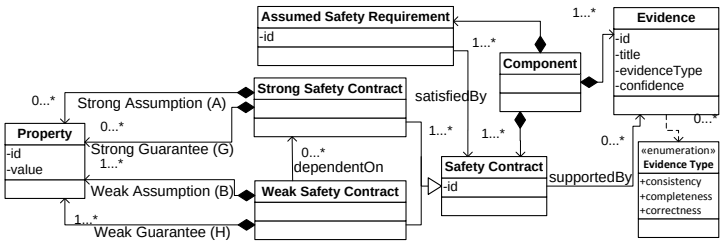

Figure 8.3: Component and safety contract meta-model

The meta-model specifies a component that is composed of safety contracts, evidence and the assumed safety requirements. Each assumed safety requirement is satisfied by at least one safety contract, and each safety contract can have supporting evidence. Additionally, we assume that there is at least one evidence provided with the component supporting the consistency of the contracts. The safety contract elements in the meta-model are covering both the strong and weak safety contracts explained in Section 8.2.2. It should be noted that, based on the SafeCer component meta-model, the components can be composite i.e., a set of interconnected subcomponents, and can represent a (sub)system.

8.3.3 Conceptual mapping of the component meta-model to GSN

As mentioned in Section 8.2.3, GSN is used for documenting safety cases by expressing arguments and supporting evidence to show that the safety claims are satisfied. At the same time, as described in Section 8.3.2, our component

Table 8.1: Conceptual mapping between the meta-model and GSN elements

The component meta-model elements	**GSN-elements**
Properties representing guarantee(s) Assumed safety requirement(s)	Goals
Evidence	Solutions
Properties representing assumption(s)	Contexts

meta-model captures the component safety claims in the safety contracts, supported by the associated evidence, with the goal to argue the satisfaction of the safety requirements. The conceptual mapping between the meta-model and GSN is depicted in Table 8.1.

In order to build an argument structure from the safety contracts, we need to map the meta-model elements to the GSN elements. Our aim is to, based on our meta-model, develop an argument-fragment that addresses the following:

1. *Compatibility of a component with a context*: to show satisfaction of strong contracts of the component by the context, as described in Section 8.2.2. Besides satisfaction, confidence in contracts needs to be addressed using associated evidence.

2. *Satisfaction of safety requirements*: to show that a safety requirement is satisfied we need to argue both, that weak contracts related to the safety requirement are satisfied, and that the set of the related contracts is sufficient to show that the requirement is satisfied.

3. *Confidence in contracts*: showing only that a contract is satisfied by a context is not enough. Evidence about confidence in the contract should be provided also. We provide evidence about confidence in contracts in terms of completeness, correctness and consistency as described in Section 8.3.1.

The satisfaction of a contract, as described in Section 8.2.2, means that the contract guarantees are offered. Consequently, properties representing the safety contract guarantees in the meta-model as well as the assumed safety requirements correspond to goals in GSN. Furthermore, we use evidence from the meta-model related to consistency, correctness and completeness as solutions within GSN. To clarify the context of our goals, we make context statements providing properties representing the assumptions of the safety contracts.

8.3.4 Overview of the architecture of the resulting argument-fragment

Given the meta-model in Section 8.3.2, we propose to generate the resulting argument-fragment based on the mapping provided in Section 8.3.3.

In the argumentation-fragment generation we will follow a pattern that for a component, say x, with a top-level goal, say $G1$, in a series of successive steps will generate the corresponding argumentation fragment. We start by decomposing the goal $G1$ into three sub-goals, as shown in Fig. 8.4. We first argue satisfaction of all the strong contracts of x in the goal $G2$. Then, we provide evidence for the consistency of all the contracts associated with x in the goal $G4$ and finally, we argue over satisfaction of the requirements by the related contracts in the goal $G3$. We now further develop the goal $G3$ and leave the goals $G2$ and $G4$ undeveloped, as they will be explored later.

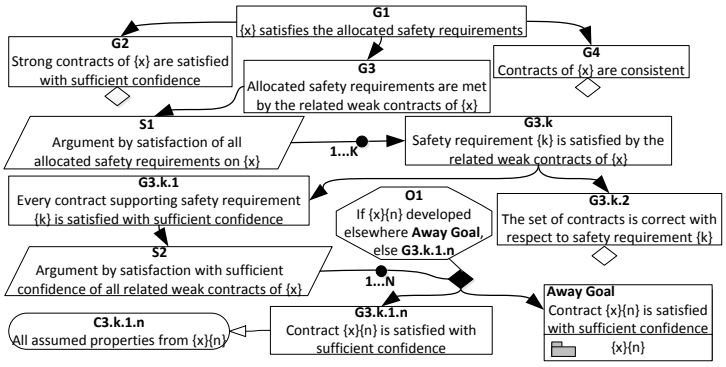

Figure 8.4: Safety requirements satisfaction goal sub-structure

We further develop the goal $G3$ by applying the strategy $S1$ to argue over satisfaction of all safety requirement allocated to component x. For every safety requirement $k \in [1, K]$ where K is the number of allocated requirements, a goal $G3.k$ is created, stating satisfaction of the requirement by the related contracts. We further break down the $G3.k$ goal into two sub-goals: (1) $G3.k.1$ arguing over satisfaction of every supporting contract of the requirement k, and (2) $G3.k.2$ providing associated evidence that the related safety contracts supporting the safety requirement k are correct with respect to the requirement. We first focus on the $G3.k.1$ goal, and leave the $G3.k.2$ goal

undeveloped, as it will be explored together with other parts of the argument referring to evidence.

When arguing over satisfaction with sufficient confidence of a set of contracts, we use the same strategy whether we argue over all the strong contracts ($G2$) or the weak contracts that support the safety requirements. To further develop the $G3.k.1$ goal, we apply the strategy $S2$ to argue over satisfaction with sufficient confidence over every related contract and reach the choice represented by obligation $O1$. If a goal has been developed elsewhere to support a contract n we create an away goal, otherwise we create a goal $G3.k.1.n$ for every contract n arguing over its satisfaction, where $n \in [1, N]$, with N being the number of related contracts to the requirement k. In order to further clarify the goal $G3.k.1.n$ we provide assumed properties of the contract n as a goal context.

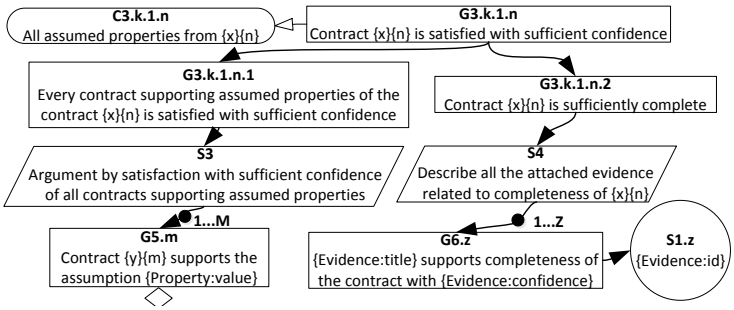

Figure 8.5: Contract satisfaction with confidence goal sub-structure

As shown in Fig. 8.5, to argue that a safety contract n is satisfied with sufficient confidence we break down the goal $G3.k.1.n$ into two sub-goals: (1) $G3.k.1.n.1$ arguing over satisfaction of every safety contract m that supports the assumed properties of the contract n, where $m \in [1, M]$ and M is the number of contracts supporting the contract n, and (2) $G3.k.1.n.2$ providing attached evidence about the completeness of contract n. We further develop the goal $G3.k.1.n.1$ by applying the strategy $S3$ to argue over satisfaction of every supporting contract m and create a sub-goal $G5.m$ arguing that the corresponding assumed property of the contract n is satisfied by the supporting contract m. To develop the goal $G5.m$ we apply the same strategy as for the goal $G3.k.1$.

For developing the three arguments that present the attached evidence re-

lated to completeness, correctness and consistency, represented by the goals $G3.k.1.n.2$, $G3.k.2$ and $G4$, we develop the argument inspired by the "Specification Argument Pattern" [13]. Unlike in that work, we define the three types of evidence differently, as described in Section 8.3.1. The goal $G3.k.1.n.2$ is developed by applying a strategy $S4$ to argue over every attached evidence of the specific type. For every evidence a goal is created claiming with what level of confidence does this goal support the completeness/consistency/correctness and the evidence reference is provided as the solution to the goal.

8.3.5 Rules for generation of component argument-fragments

Given the argument structure in Section 8.3.4 and the component meta-model we can define a sequence of transformation rules that facilitate (semi)automatic generation of argument-fragments. Our goal is not only to transfer all the information provided by the safety contracts into the argument-fragment, but also to point out the goals that need further development and thus alert safety managers. For this we use undeveloped goals within the argument-fragments. We provide the rules similarly as in [6]. We create an argument-fragment for a component x by using the following rules:

R1. Create the top-level goal $G1$: "$\{x\}$ satisfies the allocated safety requirements". Develop the goal $G1$ further by creating three sub-goals:

   (a) $G2$: "Strong contracts of $\{x\}$ are satisfied with sufficient confidence".

   (b) $G3$: "Allocated safety requirements are met by the related weak contracts of $\{x\}$".

   (c) $G4$: "Contracts of {x} are consistent".

R2. Further develop the goal $G3$ and for every allocated safety requirement k create a goal $G3.k$ "Safety requirement $\{k\}$ is satisfied by the related weak contracts of $\{x\}$" and develop this goal further by creating two sub-goals:

   (a) $G3.k.1$: "Every contract supporting safety requirement $\{k\}$ is satisfied with sufficient confidence".

   (b) $G3.k.2$: "The set of contracts is correct with respect to safety requirement $\{k\}$".

R3. Further develop the goal $G3.k.1$ by developing an argument for every safety contract n of the component x, associated with the safety requirement k. If the contract satisfaction module is developed elsewhere in the argument provide an away goal, otherwise create a sub-goal $G3.k.1.n$ "Contract $\{x\}\{n\}$ is satisfied with sufficient confidence" and provide properties representing the assumptions of the contract $\{n\}$ as the goal context $C3.k.1.n$. Further develop the sub-goal:

   (a) $G3.k.1.n.1$: "Every contract supporting assumed properties of the contract $\{x\}\{n\}$ is satisfied with sufficient confidence". For every contract m supporting the assumed property p of the contract n create a sub-goal $G5.m$: "Contract $\{y\}\{m\}$ supports the assumption $\{p\}$ ", where m is specified for a component in environment of x, say y.

   (b) $G3.k.1.n.2$: "Contract $\{x\}\{n\}$ is sufficiently complete".

R4. The goal $G5.m$ is developed further in the same way as $G3.k.1$ and the goal $G2$ is developed further in the same way as the goal $G3.k.1.n$.

R5. Goals $G3.k.1.n.2$, $G3.k.2$ and $G4$ are developed further in the same way for the list of attached evidence of the corresponding type, respectively, completeness, correctness and consistency. For every evidence z from the corresponding list of evidence type:

   (a) Create a goal $G6.z$: "$\{Evidence : title\}$ supports $\{EvidenceType\}$ of the contract with $\{Evidence : confidence\}$".

   (b) Attach a solution $S1.z$ to the goal $G6.z$ with $Evidence : id$ as reference.

R6. If no evidence of a particular type is provided, an undeveloped goal is used to indicate that the goal should be further developed.

It should be noted that, based on Rule $R4$, we can generate argument-fragments for a composite component by iterating through hierarchical structure. Applying the rules to an out-of-context component will generate an incomplete argument-fragment since not all relevant claims can be captured out-of-context. Such claims are left undeveloped, e.g., correctness of contracts with respect to a safety requirement. Hence further development of the argument-fragment is required to address all the undeveloped claims.

Table 8.2: Safety contracts for the Estimator component

A1: fuelLevelSensor within [0,5] AND fuelRate within [-1,3212] **G1**: totalFuelLevel within $[-1, 100]$ $\mathbf{E}_{A1,G1}$: Sw architecture design specification, Sw architecture verification report
B1: (fuelLevelSensor within correct range AND fuelLevelSensor does not deviate more than 10% from the actual fuel level value AND fuelLevelSensorParameter=10) OR (fuelRate within [0,3212] AND fuelRate does not deviate more than 1% from the actual engine consumption value AND Tank size within [230-1000]) **H1**: totalFuelLevel does not deviate more than -1% from the actual fuel level value $\mathbf{E}_{B1,H1}$: Simulation of the Estimator component under assumed conditions

8.4 Argument-fragment for FLES

In this section we provide safety contracts for the Estimator and EMS components of FLES, as well as show the generation of an argument-fragment for Estimator.

8.4.1 The safety contracts

The strong and weak contracts for Estimator addressing the requirement $FSR1$ of FLES are shown in Table 8.2. The strong contract assumes the allowed ranges of inputs and guarantees the possible outputs of the component. The evidence supporting the completeness of strong contract $\langle A1, G1\rangle$ includes the software architecture design specification and the corresponding verification report.

As described in Section 8.2.1, the quality of the totalFuelLevel output of the Estimator component is dependent on relevant parameters and the quality of inputs. The weak contract $\langle B1, H1\rangle$ of Estimator guarantees that the deviation of the totalFuelLevel from the actual fuel level is less than or equal to -1% if assumptions on either fuelLevelSensor and parameters related to it, or fuelRate and parameters related to it, are satisfied. The corresponding evidence is obtained by simulation of Estimator under the assumed conditions, and the simulation report is attached as evidence supporting the contract completeness.

The EMS component safety contracts related to the Estimator component are provided in Table 8.3. The EMS strong contract is similar to the one for the Estimator component, ensuring the input and output port ranges. The weak contract $\langle B2, H2\rangle$ guarantees that the deviation of the estimated fuel consump-

Table 8.3: Safety contracts for the EMS component

A2: engineStatus within [a,b]
G2: fuelRate within [-1,3212]
$\mathbf{E}_{A2,G2}$: Sw architecture design specification, Sw architecture verification report
B2: Engine parameters=20 AND engineStatus delay under 0.3 seconds
H2: fuelRate does not deviate more than 0.4% from the actual fuel consumption
$\mathbf{E}_{B2,H2}$: Simulation of the fuel consumption estimation under assumed conditions

tion does not exceed 0.4% of the actual fuel consumption under the assumed engine parameters and freshness of the information obtained from the engine. A simulation of the EMS component's behaviour under the stated conditions is attached as an evidence to support contract completeness.

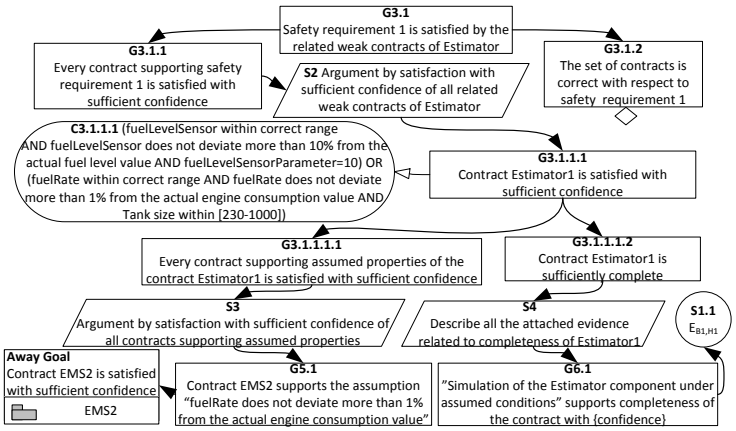

Figure 8.6: A part of the resulting argument-fragment

8.4.2 The resulting argument-fragment for the Estimator component

In Fig. 8.6 we provide a part of the argument-fragment for $FSR1$ of FLES, allocated to the Estimator component and associated with the Estimator contract $\langle B1, H1 \rangle$ denoted as $Estimator1$ within the argument. By using the rules from Section 8.3.5, we generate an argument-fragment from the provided

safety contracts to argue over satisfaction of $FSR1$ by showing that the requirement is satisfied by the related $Estimator1$ contract. The argument for satisfaction of $Estimator1$ contract is developed to show the associated evidence supporting its completeness, and point to the away goals supporting its assumed properties. Due to space limitations we show only an away goal supporting $Estimator1$ assumed property related to the fuelRate deviation and supported by the EMS $\langle B2, H2\rangle$ contract, denoted as $EMS2$ within the argument. The generated argument-fragment contains some properties that could be captured in an out-of-context setting and should be further developed to cover all relevant properties not captured within the contracts.

8.5 Discussion

As seen in the example in Section 8.4 we are able to generate a partial argument-fragment based on the component meta-model in Section 8.3.2. We support the confidence in contract completeness by associating the supporting evidence with the contracts. At the same time, by making the contracts related to the actual behaviour of the component and not to particular safety requirements, we are able to use the contracts to address different context-specific safety requirements.

The presented approach allows us to use the safety claims captured for an out-of-context component to develop context-specific argument-fragments. The resulting argument-fragment for a particular context should not include information relevant for all contexts, but only the information relevant for the particular context. By automating the generation of argument-fragments from safety contracts we speed up the creation of such argument-fragments for different contexts. The argument presented in Section 8.4 does not present all the aspects an argument should cover, such as failure modes or process-based arguments, but it provides an illustration of how the contracts can be used to generate argument-fragments. Contracts can be used to capture different safety aspects of components, e.g., failure behaviour. The resulting argument quality depends on the quality and variety (e.g., in terms of aspects) of the provided contracts.

The amount of work that still needs to be performed for a specific system depends on the abstraction level at which we allocate the safety requirements to components that have their safety contracts specified. If we connect the requirements with the contracts at higher levels of abstraction, based on the compositional nature of our approach a more complete argument-fragment

could be generated. According to ISO 26262, SEooC cannot be an item, i.e., a system implementing a complete functionality, but it can be a subsystem or a subcomponent of an item. Hence we focused on lower level components and how to reduce efforts needed to generate their argument-fragments.

The problem of automation and reuse of safety analyses and safety reasoning within the safety cases is a sensitive issue, especially since safety is a system property and needs to be reasoned about for the particular system. As mentioned in [5], the goal of automation is not to replace human reasoning, but to focus it on areas where they are best used. Similarly, in this work we are not aiming at eliminating human reasoning from the process of safety reasoning and argumentation, but to support it by providing automation of more clerical tasks.

8.6 Related Work

Generating safety case arguments to increase efficiency of safety certification has been a topic of many recent works. While some consider different notions of assumption/guarantee contracts for that purpose [14, 15] others directly build upon safety requirements [6, 7].

Assume/guarantee contracts are used in [14] to capture the vertical dependencies between a software application and a hardware platform that enables automatic generation of application specific arguments. The work presents a model-based language for specifying demanded and guaranteed requirements between the applications and platforms. The language allows for capturing restricted set of properties, whereas the contract formalism we base our work on is more expressive and offers support for easier out-of-context to in-context reuse of components. Also, [14] does not provide means for generating arguments from the captured contracts.

An approach where “informal” contracts are used for safety-case generation is proposed in [15]. The approach uses Dependency-Guarantee Relationships (DGRs) that correspond to our contracts. It derives an argument for a module by using all the DGRs of the module to build an argument relying on dependencies from other modules. In contrast to this approach, we take in consideration different types of evidence that need to be provided with the safety contracts and components, including compatibility of a component with a particular context.

A method for automated generation of safety case arguments based on an automatic extraction of information from existing work-products is presented

in [7]. The generated argumentation consists of summaries of different work-products created within a project. Similarly, a methodology for safety case assembly from artefacts required to satisfy some process objectives is presented in [6]. The work provides a set of transformation rules from captured safety requirements to safety case arguments. While these methods are useful for generating a safety case argument from a set of safety requirements that are related to existing work-products, they do not as we do consider reuse of out-of-context components developed and prepared for certification.

8.7 Conclusion and Future Work

In this paper we have presented an approach for generating safety case argument-fragments from safety contracts for out-of-context components developed and prepared for safety certification independently of the system in which they will be used. The approach allows us to speed up the creation of context-specific argument-fragments. More specifically, we have presented an overview of the argument-fragment architecture and provided a set of rules for generating the argument-fragments from the safety contracts, including illustrating the application of the rules with an example. We can conclude that safety contracts provide a good basis for generating argument-fragments and in that way allow safety engineers to focus more on capturing the knowledge about the system rather than spending time on documenting a safety case.

In our future work, we plan to refine our component meta-model, e.g., to provide support for different classes of requirements. Consequently, this refinement entails co-evolution of the generation rules. We also plan to implement the provided rules within an existing tool that supports a contract formalism, e.g., the CHESS-toolset [16]. To show the scalability of our approach we aim at using it for more complex case studies, e.g., for a larger number of safety requirements. Further more, we plan to explore how our approach could be used to reduce some of the common argument fallacies [17] related to the structure of arguments. Moreover, it is worthwhile investigating usage of our approach for safety case maintenance and change management.

Acknowledgements

Thanks to Iain Bate for useful discussions and comments. This work is supported by the Swedish Foundation for Strategic Research (SSF) via project SYNOPSIS as well as EU and Vinnova via the Artemis JTI project SafeCer.

Bibliography

[1] N. R. Storey. *Safety Critical Computer Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[2] ISO 26262-10. *Road vehicles — Functional safety — Part 10: Guideline on ISO 26262*. International Organization for Standardization, 2011.

[3] I. Bate, H. Hansson, and S. Punnekkat. Better, Faster, Cheaper, and Safer Too - Is This Really Possible? In *17th International Conference on Emerging Technologies for Factory Automation*. IEEE, September 2012.

[4] I. Sljivo, B. Gallina, J. Carlson, and H. Hansson. Strong and weak contract formalism for third-party component reuse. In *International Workshop on Software Certification*. IEEE Computer Society, November 2013.

[5] J. Rushby. Logic and Epistemology in Safety Cases. In *Proceedings of the 32nd International Conference on Computer Safety, Reliability, and Security*, volume 8153 of *LNCS*, pages 1–7. Springer-Verlag, September 2013.

[6] E. Denney and G. J. Pai. A lightweight methodology for safety case assembly. In *Proceedings of the 31st International Conference on Computer Safety, Reliability and Security*, volume 7612 of *LNCS*, pages 1–12. Springer-Verlag, September 2012.

[7] E. Armengaud. Automated Safety Case Compilation for Product-based Argumentation. In *Embedded Real Time Software and Systems*, February 2014.

[8] R. Dardar. Building a Safety Case in Compliance with ISO 26262 for Fuel Level Estimation and Display System. Master's thesis, Mälardalen Uni-

versity, School of Innovation, Design and Engineering,, Västerås, Sweden, 2013.

[9] I. Sljivo, J. Carlson, B. Gallina, and H. Hansson. Fostering Reuse within Safety-critical Component-based Systems through Fine-grained Contracts. In *International Workshop on Critical Software Component Reusability and Certification across Domains*, June 2013.

[10] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis. Multiple Viewpoint Contract-Based Specification and Design. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 5382 of *Lecture Notes in Computer Science*, pages 200–225. Springer, 2007.

[11] GSN Community Standard Version 1. Technical report, Origin Consulting (York) Limited, November 2011.

[12] J. Carlson et al. "Generic component meta-mode, Version 1.0" SafeCer, Deliverable D132, November 2013.

[13] I. Bate and P. Conmy. Assuring Safety for Component Based Software Engineering. In *15th International Symposium on High Assurance Systems Engineering*, pages 121–128. IEEE, January 2014.

[14] B. Zimmer, S. Bürklen, M. Knoop, J. Höfflinger, and M. Trapp. Vertical Safety Interfaces – Improving the Efficiency of Modular Certification. In *Computer Safety, Reliability, and Security*, pages 29–42. Springer, 2011.

[15] 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.

[16] CHESS-toolset, `http://www.chess-project.org/page/download`.

[17] W. S. Greenwell, J. Knight, C. M. Holloway, and J. J. Pease. A Taxonomy of Fallacies in System Safety Arguments. In *24th International System Safety Conference*, 2006.

Chapter 9

Paper C: A Method to Generate Reusable Safety Case Fragments from Compositional Safety Analysis

Irfan Šljivo, Barbara Gallina, Jan Carlson, Hans Hansson, Stefano Puri.
In Proceedings of the 14th International Conference on Software Reuse (ICSR 2015), Springer-Verlag, January 2015

Abstract

Safety-critical systems usually need to be accompanied by an explained and well-founded body of evidence to show that the system is acceptably safe. While reuse within such systems covers mainly code, reusing accompanying safety artefacts is limited due to a wide range of context dependencies that need to be satisfied for safety evidence to be valid in a different context. Currently the most commonly used approaches that facilitate reuse lack support for reuse of safety artefacts.

To facilitate reuse of safety artefacts we provide a method to generate reusable safety case argument-fragments that include supporting evidence related to safety analysis. The generation is performed from safety contracts that capture safety-relevant behaviour of components within assumption/guarantee pairs backed up by the supporting evidence. We illustrate our approach by applying it to an airplane wheel braking system example.

9.1 Introduction

A recent study within the US Aerospace Industry shows that reuse is more present when developing embedded systems than non-embedded systems [1]. The study reports that code is reused most of the time, followed by requirements and architectures in significantly smaller scale than code. Aerospace industry, as most other safety-critical industries, needs to follow a domain specific safety standard that requires additional artefacts to be provided alongside the code to show that the code is acceptably safe to operate in a given context. The costs of producing the verification artefacts are estimated at more than 100 USD per code line, while for highly critical applications the costs can reach up to 1000 USD [2]. In most cases, as part of the certification efforts an additional time-consuming and expensive task of providing a safety case is required. A safety case is documented in form of an explained and well-founded structured argument to clearly communicate that the system is acceptably safe to operate in a given context [3].

Most safety standards are starting to acknowledge the need for reuse, hence the latest versions of both aerospace (DO178-C) and automotive (ISO 26262) industry standards explicitly support techniques for reuse, e.g., the notion of Safety Element out of Context (SEooC) within automotive [4] and Reusable Software Components (RSC) within aerospace industry [5]. This allows for easier integration of reusable components, such as Commercial of the shelf (COTS), but it also means that some safety artefacts of the reused components should be reused as well if we are to fully benefit from the reuse and safely integrate the reused component into the new system. The difficulty that hinders reuse is that safety is a system property. This means that hazard analysis and risk assessment used to analyse what can go wrong at system level, as required by the standards, can only be performed in a context of the specific system. To overcome this difficulty compositional approaches are needed. CHESS-FLA [6] is a plugin within the CHESS toolset [7] that supports execution of Failure Logic Analysis (FLA) such as Fault Propagation and Transformation Calculus (FPTC). FPTC allows us to calculate system level behaviour given the behaviour of the individual components established in isolation. Such compositional failure analyses enable reuse of safety artefacts within safety-critical systems.

Component-based Development (CBD) is the most commonly used approach to achieve reuse within embedded systems of the aerospace industry [1]. While CBD is successfully used to support reuse of software components, it lacks means to support reuse of additional artefacts, alongside the software

components, in form of argument-fragments and supporting evidence. As a part of an overall system safety argument, argument-fragments for software components present safety reasoning used to develop a particular component and its safety-relevant behaviour, e.g., failure behaviour.

In our previous work we developed the notion of safety contracts related to software components to promote reuse of the components together with their certification data and we have proposed a (semi)automatic method to generate argument-fragments for the software components from their associated safety contracts [8]. In this work we propose a method called FLAR2SAF that uses failure logic analysis results (FLAR) to generate safety case argument-fragments (SAF). More specifically, we derive safety contracts for a component from FLAR. Then, we adapt our method for generation of argument-fragments to provide better support for reuse of the argument-fragments and the evidence they contain.

In particular, the input/output behaviour of a component developed out-of-context can be specified by FPTC rules. For example, in case of omission failure on the input I1 of the component, the component can have a safety mechanism to still provide the output O1 but with additional delay. In that case FPTC rule describing such behaviour can be specified as: $I1.omission \rightarrow O1.late$. We can use these behaviours obtained by FPTC analysis to derive safety contracts that can be further supported by evidence and used to form clear and comprehensive argument-fragments. For example, if the late failure on the output of the component can cause a hazardous event, then the corresponding argument-fragment should argue that the late failure is sufficiently handled in the context of the particular system and attach supporting evidence for that claim. For generating argument-fragments associated to the failure behaviour of the components we use an established argument pattern [9].

The main contribution of this paper is a method for the design and preparation for certification of reusable COTS-based safety-critical architectures. More specifically, we provide a conceptual mapping of FPTC rules to safety contracts. Moreover, we extend the argument-fragment generation method to generate reusable argument-fragments based on an existing argumentation pattern.

The rest of the paper is organised as follows: In Section 9.2 we provide background information. In Section 9.3 we present the rationale behind our approach and methods to derive safety contracts from FPTC analysis and generate corresponding argument-fragments. In Section 9.4 we illustrate our approach by applying it to a wheel-braking system. We present the related work in Section 9.5, and conclusions and future work in Section 9.6.

9.2 Background

In this section we briefly provide some background information on COTS-based safety-critical architectures and safety contracts. Furthermore, we recall essential information concerning the CHESS-FLA plugin within the CHESS toolset. Finally, we provide brief information on safety cases and safety case modelling.

9.2.1 COTS-based safety-critical architectures

In the context of safety critical systems, COTS-driven development is becoming more and more appealing. The typical V model that constitutes the reference model for various safety standards is being combined with the typical component-based development. As Fig.9.1 depicts, the top-down and bottom-up approach meet in the gray zone. Initially a top-down approach is carried out. The typical safety process starts with hazards identification which is conducted by analysing (brainstorming on) failure propagation, based on an initial description of the system and its possible functional architecture. If a failure at system level may lead to intolerable hazards, safety requirements are formulated, decomposed onto the architectural components, and mitigation means have to be designed. Safety requirements are assigned with Safety Integrity Levels (SILs) as a measure of quantifying risk reduction. Iteratively and incrementally the system architecture is changed until a satisfying result is achieved (i.e. no intolerable behaviour at system level). More specifically, once the safety requirements are decomposed onto components (hardware/software), COTS (developed via a bottom-up approach) can be selected to meet those requirements. If the selected components do not fully meet the requirements, some adaptations can be introduced.

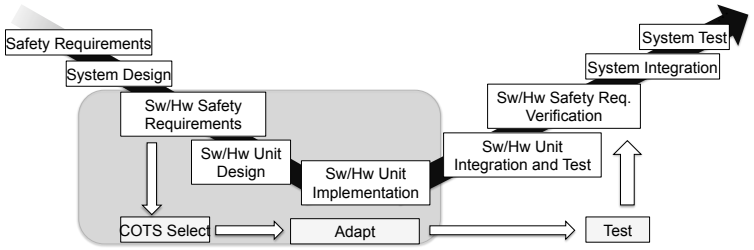

Figure 9.1: Safety-critical system development/COTS-driven development

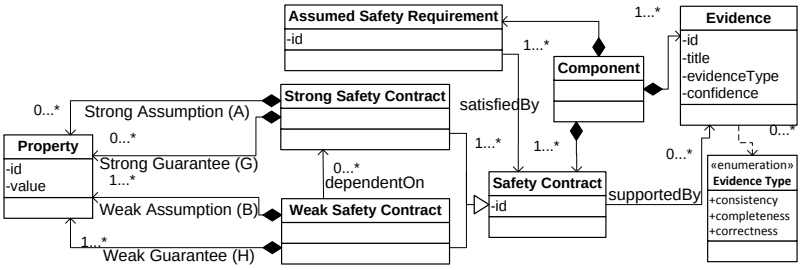

Figure 9.2: Component and safety contract meta-model [8]

To ease the selection of components, contracts play a crucial role. In our previous work, we have proposed a contract-based formalism with strong $\langle A, G\rangle$ and weak $\langle B, H\rangle$ contracts to distinguish between context-specific properties and those that must hold for all contexts [10]. A traditional component contract $C = \langle A, G\rangle$ is composed of assumptions (A) on the environment of the component and guarantees (G) that are offered by the component if the assumptions are met. The strong contract assumptions (A) are required to be satisfied in all contexts in which the component is used, hence the corresponding strong guarantees (G) are offered in all contexts in which the component can be used. For example, a strong assumption could be minimum amount of memory a component requires to operate. The weak contract guarantees (H) are offered only in those contexts where besides the strong assumptions, the corresponding weak assumptions (B) are satisfied as well. This makes the weak contracts context specific, e.g., a timing behaviour of a component on a specific platform is captured by a weak contract.

We denote a contract capturing safety-relevant behaviour as a safety contract. In [8] we introduced a component meta-model (Fig. 9.2) that connects safety contracts with supporting evidence, which provides a base for evidence artefact reuse together with the contracts. The component meta-model specifies a component in an out-of-context setting composed of safety-contracts, evidence and the assumed safety requirements. Each safety requirement is satisfied by at least one safety contract, and each contract can be supported by one or more evidence. For example, if we assume that late output failure of the component can be hazardous, then we define an assumed safety requirement that specifies that late failure should be appropriately handled. This requirement is addressed by a contract that captures in its assumptions the identified properties that need to hold for the component to guarantee that the late fail-

ure is appropriately handled. The evidence that supports the contract includes contract consistency report and analyses results used to derive the contract.

9.2.2 CHESS-FLA within the CHESS toolset

CHESS-FLA [6] is a plugin within the CHESS toolset [7] that includes two FLA techniques: (1) FPTC [11] - a compositional technique to qualitatively assess the dependability of component-based systems, and (2) FI^{4}FA [12] - FPTC extension that allows for analysis of mitigation behaviour. In this paper we limit our attention to FPTC that allows users to calculate the behaviour at system-level, based on the specification of the behaviour of individual components. In the CHESS toolset components can be modelled as component types or component implementations. Component types are more abstract and can be realised by system-specific component implementations. Component implementations inherit all behaviours of the corresponding component type.

The behaviour of the individual components is established by studying the components in isolation. This behaviour is expressed by a set of logical expressions (FPTC rules) that relate output failures (occurring on output ports) to combinations of input failures (occurring on input ports). These behaviours can be classified as: (1) a source (e.g., a component generates a failure due to internal faults), (2) a sink (e.g., a component is capable to detect and correct a failure received on the input), (3) propagational (e.g., a component propagates a failure it received on the input), and (4) transformational (e.g., a component generates a different type of failure from the input failure). Input failures are assumed to be propagated or transformed deterministically, i.e., for a combination of failures on the input, there can be only one combination of failures on the output.

The syntax supported in CHESS-FLA to specify the FPTC rules is shown in Fig. 9.3. An example of a compliant expression that demonstrates the transformational behaviour of a component is "$R1.late \rightarrow P1.valueCoarse$", which

behaviour = expression + expression = LHS '→' RHS
LHS = portname'.' bL | portname '.' bL (',' portname '.' bL) +
RHS = portname'.' bR | portname '.' bR (',' portname '.' bR) +
failure = 'early' | 'late' | 'commission' | 'omission' | 'valueSubtle' | 'valueCoarse'
bL = 'wildcard' | bR
bR = 'noFailure' | failure

Figure 9.3: FPTC syntax supported in CHESS-FLA

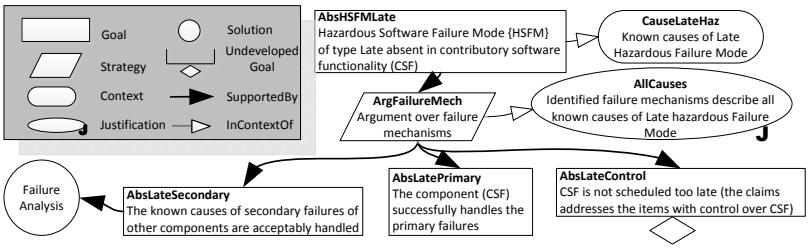

Figure 9.4: Hazardous Software Failure Mode absence pattern for type late failure

should be read as follows: if the component receives on its port R1 a late failure, it generates on its output port P1 a coarse (i.e. clearly detectable) value failure (a failure that manifests itself as a failure mode by exceeding the allowed range).

9.2.3 Safety cases and safety case modelling

A Safety case in form of an explained (argued about) and well-founded (evidence based) structured argument is often required to show that the system is acceptably safe to operate in a given context [3]. Goal Structuring Notation (GSN) is a graphical argumentation notation for documenting the safety case [13]. GSN can be used to represent the individual elements of any safety argument and the relationships between these elements. The argument usually starts with a top-level claim/goal stating absence of a failure, as in Fig. 9.4 the argument starts with a goal that has *AbsHSFMLate* identifier. The goals can be further decomposed to sub-goals with *supportedBy* relations denoting inference between goals or connecting supporting evidence with a goal. The decomposition can be described using strategy elements e.g., *ArgFailureMech* in Fig. 9.4. To define the scope and context of a goal or provide its rationale, elements such as context and justification are attached to a goal with *inContextOf* relations. For example, context *CauseLateHaz* is used to clarify the *AbsHSFMLate* goal by providing the list of known causes of the late failure mode. The undeveloped element symbol indicates elements that need further development. For more details on GSN see [13].

GSN was initially used to communicate a specific argument for a particular system. Since similar rationale exists behind specific argument-fragments

in different contexts, argument patterns of reusable reasoning are defined by generalising the specific details of a specific argument [13]. In this work we use the argument pattern for Handling of Software Failure Modes (HSFM) [9], a portion of which is shown in Fig. 9.4, to structure the generated argument-fragments related to late timing failure modes. To build an argument, HSFM pattern requires information about known causes of the failure mode and failure mechanisms that address those causes. Moreover, the failure mechanisms can be classified into three categories: (1) Primary failures within Contributory Software Functionality (CSF) that can cause the failure; (2) Secondary failures relating to other components within the system on which CSF is dependent; and (3) Failures caused by items controlling CSF e.g., in case of late hazardous failure mode the controlling item is the scheduling policy.

9.3 FLAR2SAF

In this section we present FLAR2SAF, a method to generate reusable safety case argument-fragments. We first provide the rationale of the approach in Section 9.3.1. We provide a method to translate FPTC rules into safety contracts in Section 9.3.2, and we adapt and extend the method for semi-automatic generation of argument-fragments from safety contracts in Section 9.3.3.

9.3.1 Rationale

In our work we use safety contracts to facilitate reuse of safety-relevant software components. The method for semi-automatic generation of argument-fragments from safety contracts, mentioned in Section 9.2.1, can be used to support the reuse of certification-relevant artefacts from previously specified contracts. Just as evidence needs to be provided with a reusable component to increase confidence in the component itself, similarly in some cases the trustworthiness of the evidence should be backed up as well [14]. To reuse evidence-related artefacts together with the argument fragments, additional information about the rationale linking the artefacts and the safety contracts they support should be provided. Furthermore, the issue of trustworthiness of such evidence needs to be addressed. For example, we might need to describe the competence of the engineers that performed a particular analysis or even qualification of the analysis tool.

To capture the additional information related to evidence we enrich the component meta-model presented in Section 9.2.1. We enrich the connection

between a contract and evidence by adding optional descriptive attribute capturing the rationale for how the particular evidence, or set of evidence, supports the goal. This information is used to provide additional clarification on the connection between the evidence and the claims made by the contract. Clarification of confidence in the evidence itself can be made in two different ways: either by directly including or referencing supporting information in the context of the evidence (e.g., competence of person performing the failure analysis can be found in document x); or to point to an already developed goal, called an *away goal* [13], presenting the supporting information (we could have a repository of generic argument-fragments related to staff competence and tool-qualification [15]). In the presented component meta-model we append attributes to the evidence to capture supporting information related to the evidence, including a set of references to the supporting away goals.

FLAR2SAF based on FPTC analysis can be performed by the following steps:

- Model the component architecture in CHESS-FLA;
- Specify failure behaviour of a component in isolation using FPTC rules;
- Translate the FPTC rules into corresponding safety contracts and attach FPTC analysis results as initial evidence;
- Support the contracts with additional V&V evidence and enrich the contract assumptions accordingly;
- Upon component selection, depicted in Fig. 9.1 in Section 9.2.1:
  - Perform FPTC analysis and calculate system-level failure behaviour;
  - Translate the results of FPTC analysis to system-level safety contracts;
  - Support and enrich the contracts with additional V&V evidence;
- Use the approach to semi-automatically generate an argument-fragment based on the argument pattern presented in Section 9.2.3.

The generated argument-fragment is tailored for the specific system so that only contracts satisfied in the particular system are used to form the argument, and accordingly only evidence associated to such contracts is reused to support confidence in the contracts. Particular evidence can only be reused if all the captured assumptions within the associated contract are met by the system.

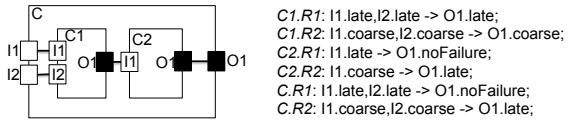

Figure 9.5: Composite component example with FPTC rules

9.3.2 Contractual interpretation of FPTC rules

In this section we focus on the step of translating the FPTC rules to safety contracts. We use the simple example in Fig. 9.5 to explain the translation process and provide a set of steps that can be used to perform the translation

In Fig. 9.5 we have FPTC rules specified for a composite component C and its subcomponents C1 and C2. When both inputs I1 and I2 exhibit late or coarse failure, component C1 acts as a propagator and outputs late/coarse failure on O1 output. Component C2 acts as a sink in case of a late failure and transforms it to no failure (e.g., a watchdog timer expires and triggers a satisfactory response), while it transforms coarse to late failure (e.g., due to additional filtering).

Safety contracts for these components can be made based on the FPTC rules. When translating the rules into contracts we consider two types of rules with respect to each failure mode: rules that describe when a failure happens (e.g., C1.R1) and rules that describe behaviours that mitigate a failure (e.g., C2.R1). We translate the first type of rules by guaranteeing with the contract that the failure described by the rule will not happen, under assumptions that the behaviour that causes the failure does not happen. The contract $\langle B, H \rangle_{C1}$ for component C1, shown in Table 9.1, guarantees that O1 will not be late if both inputs I1 and I2 never fail at the same time with late failure. This type of contracts is specified as weak since, unlike for strong contracts, their satisfaction in every context should not be mandatory. For example, in some contexts late timing failure is not hazardous, hence it is not required to be ensured.

We translate the second type of rules differently as they do not identify causes of failures, but they specify behaviours that help mitigate failures in certain cases. Since these contracts specify safety behaviour of components that should be satisfied in every context, without imposing assumptions on the environment, we denote these contracts as strong contracts. The corresponding contracts state in which cases the component guarantees that it will not exhibit any failures. We do this by guaranteeing the rule that describes this behaviour,

Table 9.1: Contracts for components C1 and C

$\mathbf{B}_{C1}$: (not (I1.late and I2.late)); $\mathbf{H}_{C1}$: not O1.late;
$\mathbf{A}_{C-1}$: -; $\mathbf{G}_{C-1}$: I1.late, I2.late $\rightarrow$ noFailure;
$\mathbf{B}_{C-2}$: (not (I1.coarse and I2.coarse)); $\mathbf{H}_{C-2}$: not O1.late;

as shown in Table 9.1 for the $\langle A, G \rangle_{C-1}$ contract for component C.

As shown on an example of translating FPTC rules from the example in Fig. 9.5 to contracts in Table 9.1, the translation can be performed in the following way for each failure:

- Identify FPTC rules that are directly related to the failure mode (either describing when it happens or describing behaviour that prevents it);
- For the rules describing when the failure mode happens:
  - Add the negation of the combination of the input failures to the contract assumptions. Connect with other assumptions with AND operator;
  - Use the absence of the failure mode as the contract guarantee;
- For the rules that describe behaviours that prevent the failure mode:
  - Use the rule within the contract guarantee to state that the component guarantees the behaviour described by the rule;

The abstract behaviour specified within the FPTC rules can be further refined so that more concrete behaviours of the component are described. For example, a refined contract related to timing failures would include concrete timing behaviour of the component in a particular context and additional assumptions related to the timing properties of the concrete system should be made.

9.3.3 Argument-fragment generation

As mentioned in Section 9.2, safety relevant components usually need to provide argument and associated evidence regarding absence of particular failures. We generate the required argument-fragment based on previously established

argument pattern HSFM for presenting absence of late failure mode, briefly recalled in Section 9.2.3. By providing means to generate context-specific argument-fragments, i.e., argument-fragments that include only information related to those contracts satisfied in the particular context, we allow for reuse of certain evidence related to the satisfied contracts.

To build an argument based on the HSFM pattern, we identify the known causes of primary and secondary failures from the corresponding FPTC rules. We identify the primary failures from the contracts translated from FPTC rules that describe behaviours that mitigate a failure mode. The secondary failures are captured within the contracts translated from FPTC rules that describe when a failure mode happens. All causes and assumptions not captured by the corresponding FPTC rules should be additionally added to the safety contracts, e.g., scheduler policy constraints. We construct the argument-fragment by using the reasoning from the HSFM pattern. The top-most goal claiming absence of the failure mode is decomposed into three sub-goals focusing on primary, secondary and controlling failures as described in Section 9.2.3. We adapt the contract-satisfaction fragment from [8] to further develop the sub-goals.

We use the safety contracts to generate the supporting sub-arguments for the primary and secondary failures and leave the goal related to controlling failures undeveloped. Supporting sub-arguments for both primary and secondary failures are generated to argue that the corresponding safety contracts are satisfied with sufficient confidence. The sufficient confidence is determined based on the specific SIL of the requirements allocated on the component and may require additional evidence in case of higher SILs. We argue the satisfaction of contracts as in [8] where we make a claim that the contract is satisfied with sufficient confidence, i.e., that the guarantee of the contract is offered. We further decompose the claim into two supporting goals: (1) an argument providing the supporting evidence for confidence in the claim in terms of completeness of the contract, and (2) an argument showing that the assumptions stated in the contract are met by the contracts of other components. We further focus on the first sub-goal related to evidence and adapt the rules related to generating the evidence sub-argument to include additionally specified information about the evidence artefacts.

For every evidence attached to a safety contract we create a sub-goal to support confidence in the corresponding safety contract. At this point we can use the additional information about the rationale connecting evidence and the safety contract and present it in form of a context statement to clarify how this particular evidence contributes to increasing confidence in the corresponding safety contract. The evidence can be further backed up by the related trust-

worthiness arguments that can be attached directly to a particular evidence. If the evidence trustworthiness information is provided in a descriptive form then additional context statements are added to the solutions, otherwise an away goal is created to point to the argument about the trustworthiness of the evidence, e.g., an argument presenting competence of a person that conducted the analysis which resulted in the corresponding evidence.

To achieve the argument-fragment generation we extended the approach for generation of argument-fragments from safety contracts [8] to allow for argument-fragment generation in the specific form of the selected pattern. The approach is adapted to generate an argument-fragment that clearly separates and argues over primary, secondary and controlling failures as described above, and to include additional information related to the evidence.

While the benefits of reusing evidence are great, a big risk can be falsely reusing evidence which may result in false confidence and potentially unsafe system. It must be noted that deriving safety contracts from safety analyses does not necessarily result in complete contracts. To increase confidence in reuse of safety artefacts, additional assumptions should be captured within the safety contracts to guarantee the specified behaviour with sufficient confidence. While this will limit reuse of the particular contract and the associated evidence, the weak safety contracts notion allows us to specify a number of alternative contracts describing particular behaviour in different contexts.

9.4 Application Example

In this section we demonstrate FLAR2SAF by applying it to a Wheel-Braking System (WBS). We first briefly introduce the WBS in Section 9.4.1. In Section 9.4.2 we apply CHESS-FLA/FPTC analysis on WBS. We use the translation steps from Section 9.3.2 to translate the contracts from the FPTC analysis results in Section 9.4.3. We present the generated argument-fragment in Section 9.4.4.

9.4.1 Wheel Braking System (WBS)

In this section we recall WBS, which was originally presented in ARP4761 [16]. We use a simplified version of WBS to illustrate the use of FLAR2SAF.

WBS is a part of an airplane braking system. It takes two input brake pedal signals that are used by the Brake System Control Unit (BSCU) to calculate the braking force. The software architecture of BSCU modelled in CHESS

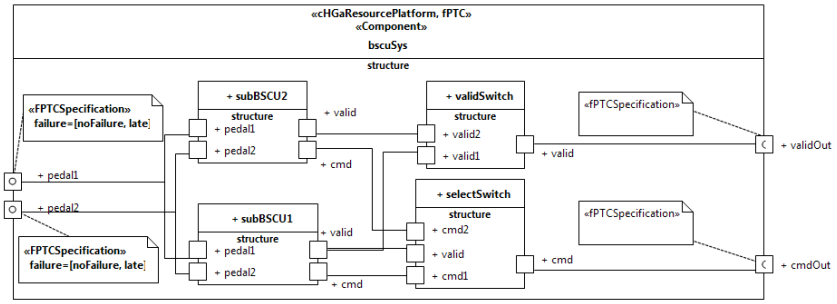

Figure 9.6: BSCU model in CHESS

Table 9.2: A subset of FPTC rules for BSCU subcomponents

Component	**FPTC rule**
subBSCU	pedal1.late, pedal2.late → valid.late, cmd.late; pedal1.noFailure, pedal2.late → valid.noFailure, cmd.omission; pedal1.late, pedal2.noFailure → valid.noFailure, cmd.omission;
validSwitch	valid1.late, valid2.late → valid.late; valid1.noFailure, valid2.late → valid.noFailure; valid1.late, valid2.noFailure → valid.noFailure;
selectSwitch	valid.late, cmd1.late,cmd2.late → cmd.late valid.noFailure, cmd1.noFailure,cmd2.late → cmd.noFailure valid.noFailure, cmd1.late,cmd2.noFailure → cmd.noFailure valid.omission, cmd1.omission,cmd2.omission → cmd.omission

is shown in Fig. 9.6. Based on the preliminary safety analysis performed on the system, the BSCU is designed with two redundant dual channel systems to meet the availability and integrity requirements. Each of the two subBSCU systems, namely subBSCU1 and subBSCU2, provide a calculated command value and a valid signal that indicates the validity of the corresponding command value. The selectSwitch forwards by default the command value from subBSCU1 if the corresponding valid signal is true, otherwise the command value from subBSCU2 is forwarded. The validSwitch component returns true if any of the signals is true, otherwise it returns false indicating that an alternate braking mode should be used, as the braking command calculated by BSCU cannot be trusted.

9.4.2 FPTC analysis

To perform the FPTC analysis we first model the system architecture in the CHESS-toolset (Fig. 9.6) and then define FPTC rules for the modelled components. The architecture and the corresponding failure behaviour of the components are defined based on the system description in Section 9.4.1.

The specified FPTC rules are shown in Table 9.2. As mentioned in Section 9.2.2, the FPTC rules specified for components are inherited by all the instances, hence the FPTC rules for the two subBSCU component implementations are the same as they are instances of the same component. The validSwitch component requires at least one valid signal present in order to forward the correct response, i.e., at least to signal that there is a problem within BSCU. Similarly, the selectSwitch component output depends both on valid and cmd signals.

As shown in Fig. 9.6 in the FPTC specifications on the input ports, we run the analysis for noFailure and late failure behaviours on the inputs. The FPTC analysis then computes the possible failures on the output ports of BSCU based on the FPTC rules for the BSCU subcomponents. The results show that the validOut port can either not fail or propagate late failures, while the cmdOut port in addition to noFailure and late failure can exhibit omission failure as well.

9.4.3 The translated contracts

The results of the FPTC analysis can be interpreted in the form of FPTC rules for the system component *bscuSys*. The resulting FPTC rule "pedal1.late, pedal2.late $\rightarrow$ validOut.late, cmdOut.late" for *bscuSys* can be translated to the contract $\langle B, H \rangle_{BSCU-1}$ shown in Table 9.4. The contract specifies that the outputs of BSCU will not be late if both input pedals are not late. The contract is supported by the FPTC analysis report from which the contract is derived.

The second translated contract $\langle A, G \rangle_{BSCU-2}$ describes the behaviour when

Table 9.3: The results of the FPTC analysis for bscuSys component

Port type	**Port label**	**Port values**
input	pedal1	noFailure, late
input	pedal2	noFailure, late
output	cmdOut	noFailure, omission, late
output	validOut	noFailure, late

Table 9.4: The translated BSCU contracts and associated evidence information

$\mathbf{B}_{BSCU-1}$: $\mathbf{H}_{BSCU-1}$:	not (pedal1.late and pedal2.late); not validOut.late and not cmdOut.late;
$\mathbf{C}_{BSCU-1}$:	The contract is derived from the FPTC analysis results for the bscuSys component;
$\mathbf{E}_{BSCU-1}$:	*name*: bscuSys FPTC analysis report *description*: FPTC analysis is performed in CHESS-toolset. *supporting argument*: FPTC_analysis_conf;
$\mathbf{A}_{BSCU-2}$: $\mathbf{G}_{BSCU-2}$:	-; pedal1.noFailure, pedal2.late $\rightarrow$ validOut.noFailure,cmdOut.omission;
$\mathbf{C}_{BSCU-2}$:	The contract is derived from the FPTC analysis results for the bscuSys component; Unit testing is used to validate that the contracts are sufficiently complete with respect to the implementation;
$\mathbf{E}_{BSCU-2}$:	*name*: bscuSys FPTC analysis report *description*: FPTC analysis is performed in CHESS-toolset. *supporting argument*: FPTC_analysis_conf; --- *name*: Unit testing results *description*: - *supporting argument*: Unit_test_conf;

only the second pedal is faulty. In that case the failure is detected by the BSCU component and reported through the validOut port, hence the validOut port reports no failure, while the cmdOut signal is omitted. The additional information related to the supporting evidence includes context statements C_{BSCU-1} and C_{BSCU-2} and a set of evidence (E_{BSCU-1} and E_{BSCU-2}). Each evidence can be further described by a context statement and supported by a set of arguments.

9.4.4 The resulting argument-fragment

A part of the resulting argument-fragment is shown in Fig. 9.7. In this argument snippet we focus only on the identified causes of primary failures (*AbsLatePrimary* goal), while the other goals shown in Fig. 9.4 remain undeveloped. We identified the BSCU-2 contract shown in Table 9.4 as the one related to primary failures as it describes behaviour of the component that mitigates a possible failure. By applying the rules to generate the contract satisfaction argument (goal *BSCU-2_sat*), we divide the argument to argue over the satisfaction of the supporting contracts (*BSCU-2_supp_sat*) and supporting evidence in con-

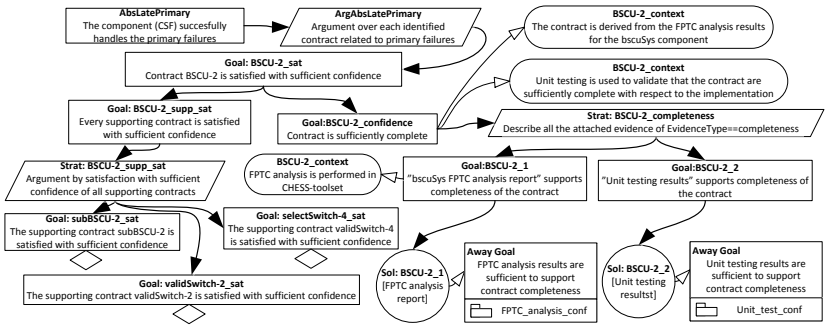

Figure 9.7: Argument-fragment based on the HSFM pattern

tract completeness (*BSCU-2_confidence*). While the argument for the *BSCU-2_supp_sat* goal follows the same pattern as for goal *BSCU-2_sat*, we focus on the argument related to the *BSCU-2_confidence* goal.

The goal *BSCU-2_confidence* is clarified by the two context statements stating that the contract has been derived from the FPTC analysis and that unit testing has been performed to validate that the contracts are sufficiently complete. In the rest of the argument we create a goal for each of the attached artefacts and enrich them with additional evidence information. The goal *BSCU-2_1* presents the confidence in the FPTC analysis. Since we do not have an argument supporting qualification of the tool used to perform the analysis we attach context statement clarifying that the FPTC analysis is performed in the CHESS-toolset. We provide an away goal related to the evidence to support trustworthiness in the analysis by arguing confidence in the FPTC analysis. Further evidence might be provided to present competences of the engineers that formed the FPTC rules and performed the analysis.

9.5 Related Work

The use of model-based development in safety-critical systems to support the development of the system safety case has been the focus of much research during the past years. Integration of model-based engineering with safety analysis to ease the development of safety cases is presented in [17]. The work presents how the architecture description language EAST-ADL2 can be used to support the development of safety-critical systems. Similarly, an approach

to handling safety concerns and constructing safety arguments within a system architectural design process is presented in [18]. The work presents a set of argument patterns and a supporting method for producing architectural safety arguments. The focus of these works is usually on extending the modelling approaches to support the safety case development process and provide guidelines on how to produce the corresponding safety arguments. Unlike in these approaches, in our work we provide a method for generating safety-arguments from the safety contracts that are based on and supported by the safety analysis performed on the system.

Deriving a safety argument from the actual source code is presented in [19]. The work focuses on constructing an argument for how the actual code complies with specific safety requirements based on the V&V artefacts. The argument skeleton is generated from a formal analysis of automatically generated code and integrates different information from heterogeneous sources into a single safety case. The skeleton argument is extended by separately specified additional information enriching the argument with explanatory elements such as contexts, assumptions, justifications etc. In contrast, in this work we generate an argument-fragment from safety contracts obtained from and supported by FPTC analysis. We utilise the contracts to specify the additional information regarding the context and additional assumptions and generate an argument-fragment for a specific failure mode covered by the FPTC analysis.

9.6 Conclusion and Future Work

Reuse within safety-critical systems is not complete without reuse of safety artefacts such as argument-fragments and the supporting evidence, since they are the key aspects of safety-critical systems development that require significant efforts. In this work we have presented a method called FLAR2SAF for generating reusable argument-fragments. This method first derives safety contracts from failure logic analysis results and then uses the contracts supported by evidence to generate reusable pattern-based argument-fragments. By an illustrative example we have shown how an argument-fragment could be generated and supporting evidence reused. The application of FLAR2SAF gives a clear indication that safety contracts can be derived from failure logic analyses. Moreover, accompanying COTS with a set of such safety contracts supported by safety evidence artefacts allows us to generate context-specific argument-fragments based on the satisfied contracts.

As our future work we are planning an evaluation of FLAR2SAF on an in-

dustrial case study. Moreover, we plan to extend the CHESS toolset to include our methods for derivation of contracts and generation of argument-fragments. We plan to explore how different types of safety analyses can be used to derive and support contracts, hence how different types of evidence could be easily reused. Another interesting future direction would be to explore how this approach can help us with change management and reuse of safety artefacts in case of changes in the system.

Acknowledgements. This work is supported by the Swedish Foundation for Strategic Research (SSF) via project SYNOPSIS as well as EU and Vinnova via the Artemis JTI project SafeCer.

Bibliography

[1] J. Varnell-Sarjeant, A. A. Andrews, and A. Stefik. Comparing Reuse Strategies: An Empirical Evaluation of Developer Views. In *International Workshop on Quality Oriented Reuse of Software*. IEEE, 2014.

[2] R. Bloomfield, J. Cazin, D. Craigen, N. Juristo, E. Kesseler, et al. Validation, Verification and Certification of Embedded Systems. Technical report, NATO, 2005.

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

[4] ISO 26262-10. *Road vehicles — Functional safety — Part 10: Guideline on ISO 26262*. International Organization for Standardization, 2011.

[5] AC 20-148. *Reusable Software Components*. FAA, 2004.

[6] B. Gallina, M. A. Javed, F. U. Muram, and S. Punnekkat. Model-driven Dependability Analysis Method for Component-based Architectures. In *Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, 2012.

[7] CHESS-toolset, `http://www.chess-project.org/page/download`.

[8] I. Sljivo, B. Gallina, J. Carlson, and H. Hansson. Generation of Safety Case Argument-Fragments from Safety Contracts. In Andrea Bondavalli and Felicita Di Giandomenico, editors, *33rd International Conference on Computer Safety, Reliability, and Security*, volume 8666 of *LNCS*, pages 170–185. Springer, Heidelberg, September 2014.

[9] R. Weaver, J. McDermid, and T. Kelly. Absence of Late Hazardous Failure Mode, `http://www.goalstructuringnotation.info/archives/218`.

[10] I. Sljivo, B. Gallina, J. Carlson, and H. Hansson. Strong and weak contract formalism for third-party component reuse. In *International Workshop on Software Certification*. IEEE Computer Society, November 2013.

[11] M. Wallace. Modular Architectural Representation and Analysis of Fault Propagation and Transformation. In *International Workshop on Formal Foundations of Embedded Software and Component-based Software Architectures*. Elsevier, 2005.

[12] B. Gallina and S. Punnekkat. FI^{4}FA: A Formalism for Incompletion, Inconsistency, Interference and Impermanence Failures Analysis. In *International workshop on Distributed Architecture modeling for Novel Component based Embedded systems*. IEEE, 2011.

[13] GSN Community Standard Version 1. Technical report, Origin Consulting (York) Limited, November 2011.

[14] R. Hawkins, I. Habli, T. Kelly, and J. McDermid. Assurance cases and prescriptive software safety certification: A comparative study. *Safety science*, 59:55–71, 2013.

[15] B. Gallina, S. Kashiyarandi, K. Zugsbrati, and A. Geven. Enabling Cross-domain Reuse of Tool Qualification Certification Artefacts. In Andrea Bondavalli, Andrea Ceccarelli, and Frank Ortmeier, editors, *International Workshop on Development, Verification and Validation of Critical Systems*, volume 8696 of *LNCS*, pages 255–266. Springer, Heidelberg, 2014.

[16] Society of Automotive Engineers (SAE) and European Organisation for Civil Aviation Equipment (EUROCAE). *ED-135/ARP-4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*. SAE, 1996.

[17] D.J. Chen, R. Johansson, H. Lönn, Y. Papadopoulos, A. Sandberg, F. Törner, and M. Törngren. Modelling Support for Design of Safety-critical Automotive Embedded Systems. In MichaelD. Harrison and Mark-Alexander Sujan, editors, *27th International Conference on Computer Safety, Reliability, and Security*, volume 5219 of *LNCS*, pages 72–85. Springer, Heidelberg, 2008.

[18] W. Wu. *Architectural Reasoning for Safety — Critical Software Applications*. PhD thesis, University of York, York, UK, 2007.

[19] N. Basir, E. Denney, and B. Fischer. Building Heterogeneous Safety Cases for Automatically Generated Code. In *Infotech@ Aerospace Conference*. AIAA, 2011.

Chapter 10

Paper D: 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, 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.

10.1 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 10.2, we present the related work and an illustrative example used to demonstrate the approach. An architecture and supporting development process, in section 10.3, 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 10.4, 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 10.5, we present a safety argument based on the use of the safety kernel and contracts. Finally, we present summary and conclusions in section 10.6.

10.2 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.

10.2.1 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 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 im-

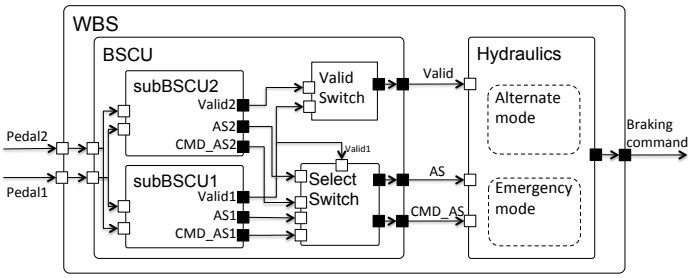

Figure 10.1: Wheel Braking System - High Level View

pact 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.

10.2.2 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. 10.1. For the purposes of this paper we consider that all six components of the WBS shown in Fig. 10.1 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.

10.3 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.

10.4 Definition of Safety Contracts

In this section we present part of the FTA performed on WBS with (section 10.4.2) and without (section 10.4.1) safety kernels. Later in section 10.4.3, 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 10.4.3 we discuss the problem of incompleteness of the safety contracts and propose how the contract completeness checking could be addressed.

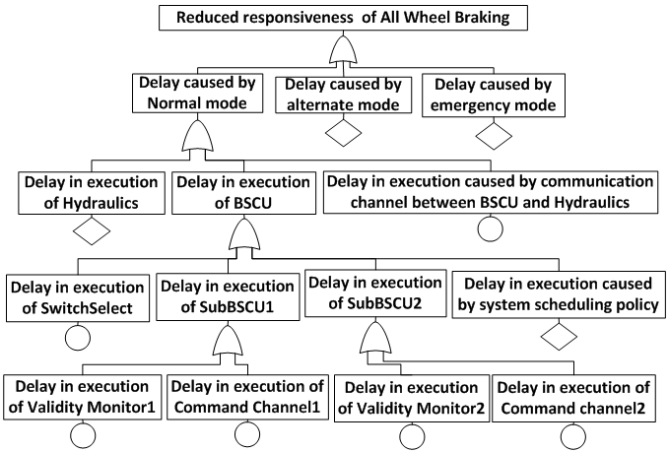

Figure 10.2: Reduced responsiveness of all wheel braking Fault Tree

10.4.1 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. 10.2 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 guaran-

tees (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:
⟨ **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 ⟩;
E1: WBS timing analysis under assumed conditions

Figure 10.3: WBS weak contract

WBS_BSCU_Weak_1:
⟨ **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 ⟩;
E2: BSCU timing analysis under assumed conditions

Figure 10.4: 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. 10.3). *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. 10.3 and 10.4 focus on the behaviour of WBS when there is no fault in the system. However the contracts don't describe behaviour

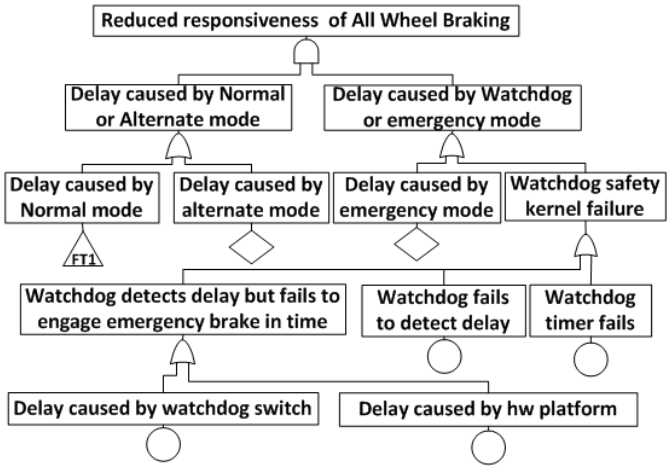

Figure 10.5: Reduced responsiveness of all wheel braking Fault Tree (updated)

of the 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.

10.4.2 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

WBS_Strong_1:
⟨ **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 ⟩;
E3: Causal analysis; Contract completeness report
WBS_Weak_1:
⟨ **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 ⟩;
E1: New WBS timing analysis under assumed conditions

Figure 10.6: WBS contracts

WBS_SKC_Strong_1:
⟨ **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 ⟩;
E4: Formal verification report

Figure 10.7: Safety Kernel strong contract

prevent 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. 10.5. 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. 10.6.

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. 10.6 and 10.7 complement the weak contracts in Figures 10.3 and 10.4 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. 10.5 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.

10.4.3 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 guarantee 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. 10.5. 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.

10.5 Safety Argument

In this section we present an overview of the graphical notation (section 10.5.1) used to construct our arguments. The WBS safety argument is presented in section 10.5.2.

10.5.1 Overview of Goal Structuring Notation

The Goal Structuring Notation (GSN) [18] – a graphical argumentation notation – explicitly represents the individual elements of any safety argument

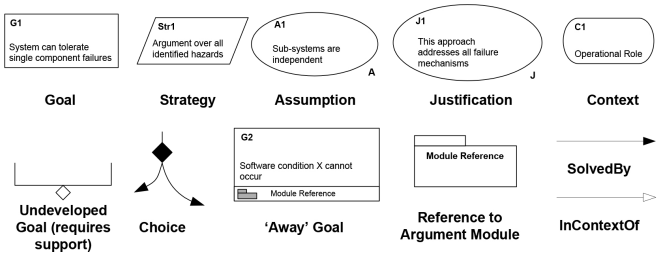

Figure 10.8: Overview of the Goal Structuring Notation (GSN)

(requirements, claims, evidence and context) and (perhaps more significantly) the 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. 10.8 (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.

10.5.2 Wheel Braking System Safety Argument

Fig. 10.9 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 10.4 and represented by

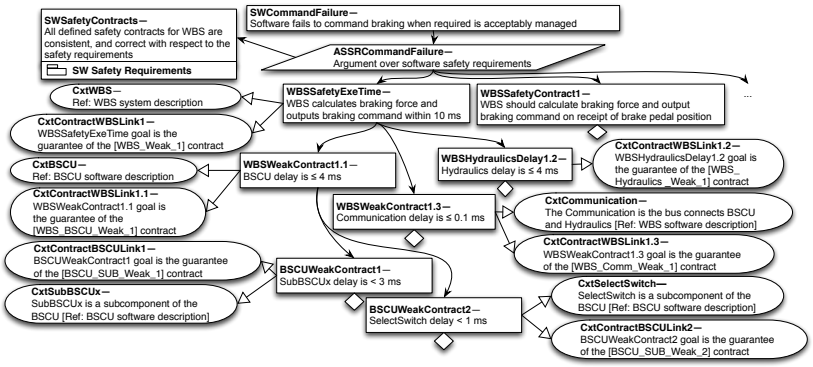

Figure 10.9: WBS safety argument before introducing the safety kernel

the goal *WBSSafetyExeTime* within the argument. We base the argument that *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-

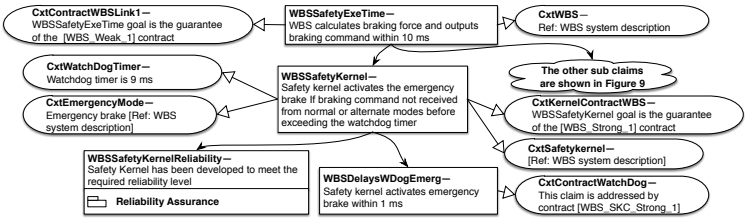

Figure 10.10: The updated WBSSafetyExeTime goal after introducing the safety kernel

responding safety argument needs to be updated as well. Based on the derived safety contracts for the safety kernel provided in Figures 10.6 and 10.7, we extend the safety argument from Fig. 10.9 with an additional supporting goal *WBSSafetyKernel* to the *WBSSafetyExeTime* claim, as shown in Fig. 10.10. The goal *WBSSafetyKernel* is clarified with context statements by referring to the corresponding contract *WBS_Strong_1* (Fig. 10.6), 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.

10.6 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] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis. Multiple Viewpoint Contract-Based Specification and Design. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 5382 of *Lecture Notes in Computer Science*, pages 200–225. Springer, 2007.

[2] W. Damm, H. Hungar, B. Josko, T. Peikenkamp, and I. Stierand. Using Contract-based Component Specifications for Virtual Integration Testing and Architecture Design. In *Design, Automation & Test in Europe Conference & Exhibition*, pages 1–6. IEEE, 2011.

[3] A. Cimatti and S. Tonetta. A Property-Based Proof System for Contract-Based Design. In Vittorio Cortellessa, Henry Muccini, and Onur Demirörs, editors, *38th Euromicro Conference on Software Engineering and Advanced Applications*, pages 21–28. IEEE Computer Society, September 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] P. Graydon and I. Bate. The Nature and Content of Safety Contracts: Challenges and Suggestions for a Way Forward. In *The 20th Pacific Rim International Symposium on Dependable Computing*. IEEE, November 2014.

[6] I. Sljivo, B. Gallina, J. Carlson, and H. Hansson. Strong and weak contract formalism for third-party component reuse. In *International Workshop on Software Certification*. IEEE Computer Society, November 2013.

[7] Modular Software Safety Case (MSSC) — Process Description. https://www.amsderisc.com/related-programmes/, November 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 *Proceedings of the 10th Annual IEEE Conference on Computer Assurance*, June 1995.

[10] B. Zimmer, S. Bürklen, M. Knoop, J. Höfflinger, and M. Trapp. Vertical Safety Interfaces – Improving the Efficiency of Modular Certification. In *Computer Safety, Reliability, and Security*, pages 29–42. Springer, 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 *Proceedings of the 10th European Dependable Computing Conference (EDCC)*, August 2014.

[13] Society of Automotive Engineers (SAE) and European Organisation for Civil Aviation Equipment (EUROCAE). *ED-135/ARP-4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*. SAE, 1996.

[14] P. Graydon and I. 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 *Proceedings of the 1997 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] J.-M. Jézéquel and B. Meyer. Design by Contract: The Lessons of Ariane. *IEEE*, 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 *Proceedings of the 21st International Conference on Computer Safety, Reliability and Security*, volume LNCS 2434, pages 321–333, 2002.

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

Chapter 11

Paper E: Using Safety Contracts to Guide the Integration of Reusable Safety Elements within ISO 26262

Irfan Šljivo, Barbara Gallina, Jan Carlson, Hans Hansson.
Technical Report, ISSN 1404-3041, ISRN MDH-MRTC-300/2015-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, March 2015
(Submitted for publication)

Abstract

Safety-critical systems usually need to be compliant with a domain-specific safety standard, which in turn requires an explained and well-founded body of evidence to show that the system is acceptably safe. To reduce the cost and time needed to achieve the standard compliance, reuse of safety elements is not sufficient without the reuse of the accompanying evidence. The difficulties with reuse of safety elements within safety-critical systems lie mainly in the nature of safety being a system property and the lack of support for systematic reuse of safety elements and their accompanying artefacts. While safety standards provide requirements and recommendations on what should be subject to reuse, guidelines on how to perform reuse are typically lacking.

We have developed a concept of strong and weak safety contracts that can be used to facilitate systematic reuse of safety elements and their accompanying artefacts. In this report we define a safety contracts development process and provide guidelines to bridge the gap between reuse and integration of reusable safety elements in the ISO 26262 safety standard. We use a real-world case for demonstration of the process, in which a safety element is developed out-of-context and reused together with its accompanying safety artefacts within two products of a construction equipment product-line.

11.1 Introduction

The basis for building modern safety-critical systems often lies in reusing existing components [1]. Most of these systems need to comply with a domain specific safety standard that often requires a safety case in form of a clear and comprehensible argument supported by evidence to show why the system is acceptably safe. The safety standards typically do not provide detailed guidelines for reusing safety elements and the accompanying artefacts, which makes the integration of the elements and the provided evidence challenging [2]. For example, the automotive safety standard ISO 26262 [3] supports reuse through the notion of Safety Elements out of Context (SEooC), which are elements explicitly developed for reuse according to ISO 26262. While the standard provides requirements and recommendations on which information is needed for the integration of SEooC, guidance on performing systematic reuse is missing.

Since safety is a system property, traditional safety analyses such as Fault Tree Analysis (FTA) and other safety artefacts such as safety arguments are made on the system level. Reusing such artefacts is difficult since what is safety relevant in one system is not necessarily safety relevant in another system. Non-systematic reuse of safety artefacts has shown to be dangerous [4], hence there is a need to fill the gap created by the safety standards' lack of guidelines for systematic reuse and integration of safety elements and their safety artefacts.

Systematic reuse of safety artefacts can be achieved by generative reuse. The term "generative reuse" is used to indicate indirect reuse of artefacts [5], be it the code itself, results of a failure analysis such as FTA [6] or parts of safety arguments [7], where a customised artefact is generated for a specific context from specification written in a domain specific specification language. For example, an out-of-context component with a pre-developed safety argument is reused in a particular system. Such safety argument, produced out-of-context, might contain irrelevant information for the particular system. Instead of trying to reuse and integrate pre-developed safety arguments, the relevant information for the particular system is first identified from the provided artefacts, and then the corresponding system safety argument is generated from the identified information. In our work we use safety contracts for capturing safety-related information and for identifying the relevant information for a particular environment.

A contract is an assumption/guarantee pair, where a component offers guarantees about its own behaviour if the assumptions on its environment are met. Safety contracts are a specific types of contracts that deal specifically with

component behaviours that are deemed relevant from the perspective of hazard analysis. In our previous work we showed how safety contracts can be used to support generative reuse of safety artefacts [8]. Since reusable elements can exhibit different behaviours in different environments, contracts are characterised as either strong or weak to allow capturing these different behaviours in a more flexible manner [9]. Furthermore, since the safety contracts deal with some of the information used in the safety arguments, we can use the contracts to semi-automatically generate the argument-fragments related to components [10].

In this report we complement the safety guidelines provided by ISO 26262 to include contract-specific activities and facilitate systematic reuse that aims at easing integration of safety-relevant components and the provided evidence within ISO 26262 systems. We first define the safety contracts development process and the corresponding contract-specific activities. Then we provide guidelines on how and when to use the contract-specific activities in the case of SEooC.

To demonstrate the proposed process we use a product-line scenario as a common real-world case. The product-line is composed of two construction machines whose compliance to ISO 26262 will be required in the near future. Both machines are equipped with lifting arms, whose software controller in both cases includes a component for automatic positioning of the arm in a predefined position. We develop this component as a SEooC and then reuse it within the two products. On this real-world case we demonstrate how safety contracts can be used for SEooC development. Moreover, we illustrate the benefit of generative reuse of safety arguments on the the SEooC integration within the two products.

The contributions of this work are (1) the guidelines in form of safety contracts development process describing the role of the safety contracts within the development and integration of reusable components within safety-critical systems, (2) alignment of the proposed process with the ISO 26262 safety process, and (3) its demonstration in a real-world case. In contrast to existing works that focus on facilitating reuse of safety artefacts within safety-critical systems [11, 12, 13, 14], we focus on detailing the guidelines for development and integration of reusable safety elements within safety-critical systems via safety contracts. More specifically, we align the proposed process with ISO 26262 to facilitate generative reuse of safety artefacts, primarily safety arguments. We focus on providing means for capturing the SEooC assumptions recommended by the standard and support their validation during integration of the SEooC in an ISO 26262 compliant system. We assume that for the integration to work, both the SEooC and the target ISO 26262 system have safety

contracts established.

The rest of the paper is structured as follows: In Section 11.2 we provide background information. We present the safety contracts development process and align it with the SEooC development process recommended by ISO 26262 in Section 11.3. In Section 11.4 we demonstrate the proposed process and the related guidelines on a real-world case. We provide discussion in Section 11.5 and related work in Section 11.6. Finally, conclusions and future work are presented in Section 11.7.

11.2 Background

In this section we provide background information on the ISO 26262 safety process and the processes recommended for development and integration of Safety Elements out of Context. Furthermore, we provide essential information on the strong and weak safety contracts as well as graphical argumentation notation for representing safety arguments.

11.2.1 ISO 26262

ISO 26262 [3] has been developed as a guidance to provide assurance that any unreasonable residual risks due to malfunctioning of E/E systems have been avoided. The standard requires a safety case in form of a clear and comprehensible argument to show that the safety requirements allocated to an item are complete and satisfied by the evidence generated during the system development. An *item* within ISO 26262 is composed of at least a sensor, controller and an actuator, which together implement a function at the vehicle level.

Central part of Fig. 11.3 shows the safety process of the ISO 26262 standard. The process starts with the *Concept phase* (Part 3 of the standard) that is initiated with the *item definition* activity where the main objective is to define and describe the item by capturing its dependencies on, and interactions with, its environment. In the subsequent activities of this phase the hazards related to the item are identified and classified according to Automotive Safety Integrity Levels (ASILs), safety goals are established and further refined into functional safety requirements that are allocated to the architectural elements.

In the first part of the *Product development at system level* phase, the technical safety requirements are derived from the functional safety concept, and the system is designed to comply with both the technical and functional safety requirements. Based on the system design, development and testing of both

the hardware (HW) and software (SW) elements is performed. During *Product development at HW/SW level* (Parts 5&6 shown in Fig. 11.3) the corresponding HW/SW safety requirements are derived with consideration of environmental/operational constrains identified during the concept phase. The process continues with integration and testing of the HW/SW elements, followed by integration of elements that compose an item to form a complete system, and then the item is integrated with other systems and tested on the vehicle level. The *Product development at system level* is finalised with safety validation and an assurance case is presented to show that the safety goals are sufficient and that they have been achieved.

We provide additional information on the concept and system design phases as they play an important role in reuse of safety elements. Based on the ISO 26262 development process, the information that needs to be gathered during these phases includes the following: (1) purpose and functionality of the item, (2) operating modes and states of the item (including the configuration parameters), (3) law, regulation and standard requirements, (4) operational and environmental constraints, (5) interface definition, (6) hazard analysis results, including the known hazards, their ASILs and the associated safety goals.

To ease the development of ISO 26262 compliant systems, the standard acknowledges different reuse scenarios: (1) elements that have been developed for reuse according to ISO 26262 in form of SEooC, (2) pre-existing elements not necessarily developed for reuse or according to ISO 26262 that have to be qualified for integration, and (3) elements that qualify for reuse as proven-in-use. In this report we focus on the SEooC reuse scenario.

Safety Element out of Context

SEooC can be an element used to compose an item, but it cannot be an item since item implements functions at vehicle level, while a reusable elements such as SEooC are not developed in the context of a particular vehicle. The development of SEooC follows the ISO 26262 safety process, but since SEooC is developed out-of-context, the information related to the system context (gathered during the concept and system design phases) first needs to be assumed. The assumptions are made to the functional safety concept as the main output of the concept phase and the external design (system-level assumptions; the interactions with, and dependencies on the elements in the environment are assumed). After assuming the relevant system design, the development of the SEooC follows the product development at SW/HW level.

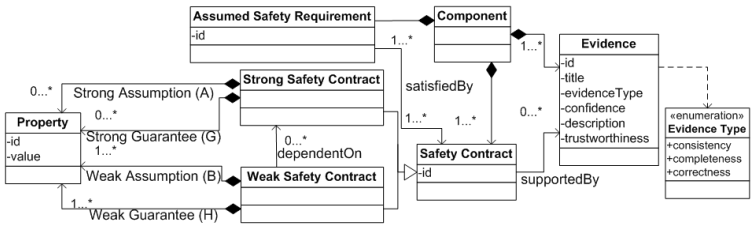

Figure 11.1: Component and safety contract meta-model

11.2.2 Safety Contracts

In our previous work [9], we have proposed a contract-based formalism with strong $\langle A, G\rangle$ and weak $\langle B, H\rangle$ contracts to distinguish between context-specific properties and those that must hold for all contexts. A traditional component contract $C = \langle A, G\rangle$ is composed of assumptions (A) on the environment of the component and guarantees (G) that are offered by the component if the assumptions are met. The strong contracts capture behaviours of components that should always be guaranteed (strong guarantees G) and the corresponding strong assumptions (A) that should always be met. The weak contracts handle behaviours that are not required to hold in every environment (weak guarantees H), but only when besides all the strong assumptions, the corresponding weak assumptions (B) are satisfied as well. For example, strong contracts can be used to prevent misuse of configuration parameters of the component by requiring parameters scope and guaranteeing interaction of the different parameters, while weak contracts could be used to describe distinct component behaviours achieved by the different configurable parameter values. The *related contracts* of a contract *C* are those contracts that either assume the guaranteed properties of *C* or the ones which guarantee properties that are assumed by the contract *C*.

As introduced in Section 11.1, we call a contract capturing safety-relevant behaviour a *safety contract*. Since not all safety-relevant information can be captured in formal contracts, we recognise that contracts consist of both formal and informal assumptions and guarantees. The formal part of the contracts can for example be captured in form of Failure Propagation and Transformation Calculus (FPTC) syntax [8].

The component meta-model (Fig. 11.1) that connects safety contracts with supporting evidence provides a base for evidence reuse together with the contracts [10, 8]. The component meta-model specifies a component in an out-

of-context setting, composed of safety-contracts, evidence and the assumed safety requirements. Each safety requirement is satisfied by at least one safety contract, and each contract can be supported by one or more evidence. This component meta-model is used as the basis for semi-automatic generation of safety case argument-fragments [10]. For example, if we assume that late output failure of the component can be hazardous, then we define an assumed safety requirement that specifies that late failure should be appropriately handled. This requirement is addressed by a contract that captures in its assumptions the identified properties that need to hold for the component to guarantee that the late failure is appropriately handled. The evidence that supports the contract includes the contract consistency report and analysis results used to derive the contract.

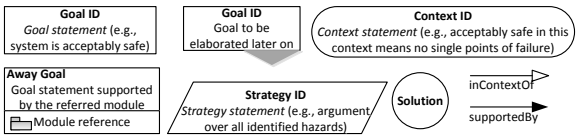

Figure 11.2: A subset of GSN symbols used within this article

11.2.3 Overview of Goal Structuring Notation

The Goal Structuring Notation (GSN) [15] is a graphical argumentation notation that can be used to represent the individual elements (e.g., goals/claims, evidence, context) of any safety argument. More importantlynly, GSN can be used to capture the relationships that exist between the individual elements by using the two relationships *inContext* and *supportedBy*. The *inContext* relationship connects claims with the contexts that are used as the clarifications of the related claims, while the *supportedBy* relationship is used for connecting goals with its subgoals, backing up goals with evidence and specifying the decomposition strategies used to decompose a goal to a set of subgoals. Basic symbols of GSN used in this article are shown in Figure 11.2 (with examples for each of the elements).

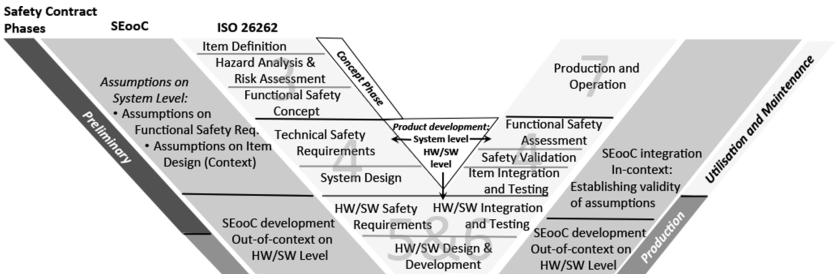

Figure 11.3: ISO 26262, SEooC and safety contract development phases mapping

11.3 ISO 26262 Safety Process Supported by Safety Contracts Development Process

In this section we present the guidelines for using the strong and weak safety contracts for development and integration of reusable safety elements within safety-critical systems. Moreover, we align the guidelines with the ISO 26262 safety process. More specifically, we first define the safety contracts development process and the contract-specific activities, and then we detail how and when these activities can be aligned with the SEooC development process.

11.3.1 Safety Contracts Development Process

As mentioned in Section 11.1, the nature of safety being system property and the dangers of non-systematic reuse hinder reuse of safety elements within safety-critical systems. To alleviate these issues a clear process and guidelines on how to perform reuse should be provided to promote systematic reuse of safety elements. To integrate the systematic reuse approach based on strong and weak safety contracts within a safety process, a safety contracts development process needs to be defined. We propose such a process divided into three phases: (1) *Preliminary* safety contracts, (2) Safety contracts *production*, and (3) Safety Contract *utilisation and maintenance*. The alignment of the safety contract, SEooC and ISO 26262 development phases is shown in Fig. 11.3. In the reminder of this subsection we provide more details about the corresponding contract-specific activities each phase is constituted of.

Preliminary Safety Contracts Phase

- *Establishing strong and weak contracts*: The strong contracts are established by considering behaviours such as nominal functional or safety mitigation behaviours not bound to context-specific configuration parameters. In contrast, weak contracts are established by considering behaviours bound to context-specific configuration parameters (e.g., accuracy of an algorithm may depend on the physical properties of the system in which it is used).

- *Enriching assumptions with environmental/operational constrains*: The different types of properties that should be captured by safety contracts include nominal functional behaviour, failure logic behaviour, resource usage behaviour and timing behaviour [14]. Upon establishing the strong and weak contracts, the contract assumptions need to be enriched to achieve sufficient level of completeness by including environmental properties such as platform properties, HW/SW interface and/or dependencies to other elements.

- *Preliminary matching of contracts to HW/SW safety requirements*: As mentioned in Section 11.2.2, the safety contracts should capture information needed to satisfy the safety requirements allocated to the corresponding safety element. For example, supporting each derived SW safety requirement allocated to a software element with at least one preliminary contract is the final goal in completing the set of the preliminary safety contracts. If the contract to satisfy a particular requirement has not been previously developed, a preliminary contract should be established with its guarantee reflecting the corresponding requirement.

Safety Contracts Production Phase

- *Actualisation of the contracts with implementation-specific properties*: Since not all information is fully known during the preliminary safety contracts phase, certain preliminary contracts (e.g., on resource usage) can only be captured with speculative targeted behaviour. After the product development stage, such contracts need to be finalised once the actual behaviour of the element (or a more accurate approximation) can be established. For example, when more accurate information about the actual accuracy of an algorithm, actual timing behaviour, or actual memory footprint of the element is available, then we can actualise the contracts

capturing such behaviours with the actual implementation-specific values.

- *Supporting contracts with evidence*: The final step in producing the safety contracts for reuse is to support such contracts with the evidence supporting the information captured by the contracts. For example, in case that information captured within a safety contract is based on simulation or testing results, the corresponding guarantee of the contract should be based on the results while the assumptions should capture the environmental parameters under which the simulation/testing has been performed. The artefacts related to the simulation/testing are then attached to the particular safety contract with a description in which way they are related. Further trustworthiness evidence can be attached to the artefacts [8]. Since each safety requirement is associated with an ASIL, which in turn influences the stringency of evidence that needs to be provided to assure that the particular requirement is satisfied, the achieved ASIL information is attached to the evidence rather than to the contracts themselves. In this way the safety requirements are connected to the achieved ASILs through the connection of the safety contracts with the associated evidence.

Utilisation and Maintenance Phase

- *Contract-based verification*: The results of unsuccessful verification can be interpreted as follows:
  - *Contracts are contradicting each other* (e.g., strong and weak contracts of the same component make contradicting assumptions on the same property). To address this result, either the existing contract assumptions and guarantees should be re-established or a replacement component should be used instead.
  - *Not all strong or relevant weak contract assumptions are satisfied*, which means that the component is either not compatible with the particular context or cannot satisfy a particular safety requirement allocated to it. In either case, the system can be re-designed to handle the incompatibility (e.g., by adding component wrappers to convert the input/output signals to a compatible format) or the reused component itself can be replaced or modified.
  - *Contract assumptions are incomplete*, e.g., safety contract on timing behaviour of component X guarantees the timing behaviour

based on platform assumptions (including compiler configuration), while a related contract on timing behaviour of component Y does not include assumptions on compiler configuration. In this case the contracts with missing assumptions should be re-evaluated and either enriched with the appropriate assumptions or a new contract should be established to capture the newly identified behaviour.

- *Contract maintenance*: In case of changes to the existing contracts, all contracts of the corresponding component should be revisited, while when updating contracts with additional assumptions, only contracts capturing the same type of behaviour (e.g., timing) should be reassessed. Modifications of a component or system design requires that all its contracts are reassessed and reestablished if required.

- *Contract-based artefacts generation*: In this article we focus on the utilisation of contracts for safety case argument generation. The generative reuse potential of the contracts can be utilised for generating other artefacts, but that is out of scope of this article.

11.3.2 SEooC Development with Safety Contracts

SEooC development starts by capturing the system-level assumptions as shown in Fig. 11.3. Simultaneously, the preliminary safety contracts phase is initiated, as described in the Section 11.2.1. All relevant assumed properties should be covered by the established preliminary contract assumptions. Once the HW/SW safety requirements are derived, each requirement is associated with at least one contract such that the behaviour achieved by the associated contracts satisfies the required behaviour by the corresponding requirement. After the safety contracts are established and associated with the safety requirements, the safety contract production phase and the corresponding ISO 26262 product development at HW/SW level are continued to develop the SEooC and its safety contracts. At this point the development of the SEooC out-of-context is completed.

Once the SEooC is used in a particular system (in-context), the assumed requirements are compared and matched to the actual safety requirements allocated to the element, and contracts are used to verify that the assumptions captured during the SEooC development are satisfied. The contract production phase continues in-context of a particular system to capture the behaviours of the SEooC that could not be established out-of-context. In case of assumptions

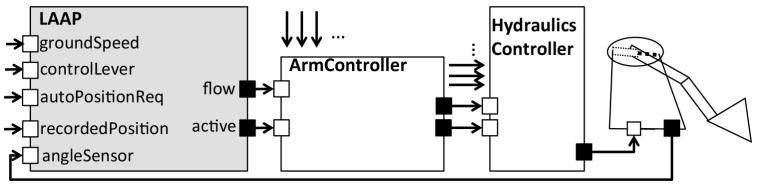

Figure 11.4: The assumed structure of the lifting arm unit context

mismatch, ISO 26262 impact analysis can be assisted by the contract maintenance activity. Once all the relevant safety contracts are satisfied for the reused SEooC, an argument for the element is generated to show the satisfaction of the safety requirements through the satisfaction of the associated safety contracts [10].

11.4 Real-world Case

In this section we demonstrate the application of the guidelines introduced in Section 11.3. We use a product-line based case for our demonstration as it is a common scenario found in industry. The aim of the case is to develop a component out of context of a particular product and reuse the component and its accompanying artefacts in two different products of a product-line. This can be challenging even for two similar products such as those in a product-line since the hazards related to the products can differ, as discussed in Section 11.1. The SEooC we develop is a Lifting Arm Automatic Positioning (LAAP) component commonly used within wheel-loaders and other construction machines. We first present the LAAP and its SEooC development, and then we discuss the LAAP integration within two different products of a wheel-loader product-line.

11.4.1 SEooC definition and development

As discussed in Section 11.3.2, the development of a SEooC starts by making assumptions on the item in which the component is intended to be used. The assumed structure of the lifting arm unit context for a wheel-loader is shown in Fig. 11.4. Wheel loaders are equipped with a lifting arm, which can perform up and down movements that are directly controlled by a hydraulic controller. The operator controls of interest for the development of the LAAP consist of a

control lever that is used to lift/lower the arm and an automatic position request button that positions the arm in a predefined position. Once the automatic positioning is started, it can be stopped by moving the control lever and switching automatically to manual mode. Besides the operator controls, the LAAP uses an arm angle sensor to determine the current arm position, recorded position to which the arm should be moved and the ground speed of the vehicle for tracking the vehicle movements. The assumptions include only information deemed relevant to the SEooC development, hence the full interface of the arm controller is not assumed at this stage.

Before specifying the assumed software safety requirements that the LAAP will implement, we need to assume safety implications of the component and its relation to possible hazards. We identified contributions of LAAP to two possible vehicle-level hazards: *(H1) unintended movement of the lifting arm*, and *(H2) hydraulic leakage*. We consider the hazards in the following operational situations:

- high speed (the vehicle is moving with varying speeds that can go up to the maximum available speed)
- short cycle (a combination of load lifting and low speed transportation)
- load and carry (the vehicle is moving with varying ground speed with bucket fully loaded)

Hazard H1 can be dangerous during high speed due to e.g., heavy traffic when driving on a public road, during the short cycle and load and carry phases it can be dangerous to bystanders present in the area while high precision movement is required from the machine. LAAP can contribute to hazard H1 by e.g., value failure of the flow command that can be caused by value failures of the angle sensor and the recorded position variable. Furthermore, the unintended arm movement can occur in case of omission of the autoPositionReq signal. Omission or late failure of the control lever signal can cause LAAP to continue its operation when not intended.

The high-pressure hydraulic leakage could produce a highly flammable oil/air mixture spray mist that might ignite in contact with hot surface, hence the leakage should be identified as soon as possible. One way in which LAAP can contribute to this situation is when the LAAP starts operating but due to the leakage the arm either never reaches the recorded position or it moves much slower than usual, which contributes to increasing the leakage. The occurrence of the hazard H2 in either of the operational situations can be danger to the

Table 11.1: SW Safety Requirements

SWSR1	Safe state shall be applied during high-speed	ASIL B
SWSR2	The stop position of the arm shall not deviate more than $\pm$ 0.04 rad	ASIL B
SWSR3	Safe state shall be applied if erroneous input (ground speed, angle sensor, control lever or recorded position) is detected	ASIL B
SWSR4	Safe state shall be applied if the operational time of the LAAP is taking more than the maximum raise time of the lifting arm	ASIL A
SWSR5	LAAP shall not start inadvertently	ASIL B
SWSR6	Safe state shall be applied when manual arm movement is in progress (i.e., when control lever value not 0)	ASIL B

driver, other participants in traffic and bystanders present in the area. To address these possible hazardous events related to both hazards, functional safety concept is assumed and the corresponding software safety requirements are derived (Table 11.1).

The strong and weak contracts of the LAAP, initially captured during the *Preliminary Safety Contracts* phase to address the SW safety requirements, are shown in Table 11.2. The strong contract *LAAP-1* requires that the groundSpeedLimit is set below 20km/h and guarantees that LAAP will be disabled when the ground speed of the vehicle is greater than the groundSpeedLimit parameter. Disabling of the LAAP is the safe state achieved by setting the active flag to false and the flow value to 0.

The strong contract *LAAP-2* specifies the assumed value ranges of the input signals and guarantees that the safe state shall be applied when either of the inputs is out of bounds. In case of *controlLever* signal, the LAAP can be active only when the lever is inactive (i.e., when the lever is 0), hence the contract specifies that when *controlLever* is different than 0, the safe state shall be applied.

The strong contract *LAAP-3* describes a SW watchdog timer implemented as a part of the component that disables LAAP if its operation time is longer than expected. To detect possible hydraulic leakage, the timer is set within the interval bound by *raiseTime* parameter that represents the maximum lifting time of the arm under full load from lowest to highest position.

The weak contracts *LAAP-4* and *LAAP-5* capture failure propagation behaviour of the LAAP such that they state which conditions should the environment of the LAAP fulfil to mitigate a potentially hazardous failure propagation.

Table 11.2: LAAP Safety Contracts

$\mathbf{A}_{LAAP-1}$:	*groundSpeedLimit* within [0, 20] km/h AND *groundSpeed* within [0, 200] km/h;
$\mathbf{G}_{LAAP-1}$:	*groundSpeed* $>$ *groundSpeedLimit* **implies** (*active* = false and *flow* = 0)
$\mathbf{A}_{LAAP-2}$:	*groundSpeed* within [0, 200] km/h AND *angleSensor* within [0,3] rad AND *controlLever* within $\pm$ 1 rad AND *recordedPosition* within [0,3] rad;
$\mathbf{G}_{LAAP-2}$:	(*groundSpeed* not within [0, 200] km/h OR *angleSensor* not within [0,3] rad OR *controlLever* not 0 rad OR *recordedPosition* not within [0,3] rad;) **implies** (*active* = false and *flow* = 0)
$\mathbf{A}_{LAAP-3}$:	*watchdogTimerInterval* within [*raiseTime*, 1.2**raiseTime*] AND *raiseTime* $>$ 0;
$\mathbf{G}_{LAAP-3}$:	(not (*active* = false and *flow* = 0) **implies** watchdogTimer start) AND (*LAAP-OperationalTime* $>$ *watchdogTimerInterval* **implies** (*active* = false and *flow* = 0 and watchdogTimer reset));
$\mathbf{B}_{LAAP-4}$:	not *angleSensor.valueFailure* AND not *recordedPosition.valueFailure*;
$\mathbf{H}_{LAAP-4}$:	not *flow.valueFailure*;
$\mathbf{B}_{LAAP-5}$:	not *autoPositionReq.comission* AND not *controlLever.omission*;
$\mathbf{H}_{LAAP-5}$:	not *flow.comission* AND not *active.comission*;
$\mathbf{B}_{LAAP-6}$:	*angleSensor* accuracy is 0.02 rad AND actuation deviation is within $\pm$0.01 rad AND *recordedPosition* does not introduce deviation;
$\mathbf{H}_{LAAP-6}$:	*flow* accuracy is 0.01 rad **implies** stop position is within $\pm$0.04 rad from the *recordedPosition*

The *LAAP-4* contract specifies that in order to avoid the flow command value failure, the environment of the LAAP should guarantee that the angle sensor signal and recorded position value do not exhibit value failure. The *LAAP-5* contract contract specifies that in order to mitigate inadvertent commands sent from the LAAP (in form of commission failures of the *flow* and *active* output ports), the environment should ensure that commission of the *autoPositionReq* signal and omission of the *controlLever* signal do not occur.

The weak contract *LAAP-6* relates the guaranteed *flow* accuracy and the lifting arm stop position based on the assumptions on the accuracy of the angle sensor, recorded position and the actuation.

The matching of the established contracts and the SW safety requirements

Table 11.3: SW Safety Requirements and safety contracts mapping

SWSR1	*LAAP-1*
SWSR2	*LAAP-4*, *LAAP-6*
SWSR3	*LAAP-2*
SWSR4	*LAAP-3*
SWSR5	*LAAP-5*
SWSR6	*LAAP-2*, *LAAP-5*

is presented in Table 11.3. The contract *LAAP-4* is not fully addressing the requirement *SWSR2*, but it only establishes that the accuracy of the flow command is dependent on the accuracy of the angle sensor and the recorded position value. Hence a more concrete contract *LAAP-6* is established to fully address the requirement *SWSR2*. During the *Safety Contracts Production* phase, the contract *LAAP-6* is updated with the actual accuracy of the *flow* command.

As mentioned in Section 11.3, the SW safety requirements addressed by the safety contracts are supported with evidence through the connection of the contracts and the supporting evidence. Since requirements are categorised with ASILs, the stringency of the evidence supporting the contracts should be appropriate for the corresponding integrity level. Since the assumed requirements are associated with at most ASIL B, to support the contracts associated with the requirements we use inspection and testing as verification means recommended by ISO 26262 for the specified ASILs. The context statements that provide clarifications of the contracts and the supporting evidence attached during Safety Contract Production phase are shown in Figure 11.4. The context statements are denoted with *LAAP-x_Cy* and evidence with *LAAP-x_Cy.*, where *x* is the number of the related contract and *y* the number of the evidence/context statement.

11.4.2 SEooC Integration

The two products in which we reuse the developed SEooC are a part of the same wheel-loader product line. First product is a Gigant Wheel-loader (GWL) used within closed construction sites. Due to its size, both the GWL itself and its arm move slower than other machines. Time needed to raise the arm under full load from minimum to maximum position is around 10 seconds. The second product is a Small Wheel-loader (SWL) used for less intensive tasks and often outside of construction sites (e.g., public service). It is much more compact than the GWL and it has two times faster lifting arm raise time.

Due to the differences between the two products, what is hazardous in one product is not necessarily hazardous in the other. Since the GWL is used in a controlled environment and its tasks do not require high precision, the value failure of the LAAPs' flow port is not considered hazardous in that case. Hence, the requirement *SWSR2* is not considered safety-relevant in context of the GWL, but is regarded as quality management. Moreover, the weak contracts *LAAP-4* and *LAAP-6* are not satisfied in the context of GWL, as integrity of the sensor data and recorded position is not ensured for the *LAAP-4* contract, and the assumption on actuation accuracy for the *LAAP-6* contract.

In contrast to the GWL, since the SWL is used in less controlled environments for tasks that usually require precision, the LAAP accuracy is much more critical. Besides a higher quality angle sensor to ensure high confidence in sufficient accuracy of the *angleSensor* input to the LAAP, an error-detecting code is used to ensure that the stored *recordedPosition* has not been accidentally changed (e.g., due to bit flip). Contracts of the corresponding components guarantee these properties of the angle sensor and the *recordedPosition* variable which satisfies the contract *LAAP-6*, while the contract *LAAP-4* is not satisfied in the SWL system either as it would be too expensive to achieve it.

Since the strong contract *LAAP-1* requires *groundSpeedLimit* to be set in every vehicle to a value below 20 km/h, both products must set the appropriate values. In the GWL the limit is 20 km/h, since the arm moves much slower and in a controlled environment, while the limit is 10 km/h for the SWL.

Once the reused contracts are checked and new contracts established during the*Utilisation and Maintenance* phase, we utilise the contracts for the generation of safety argument-fragments. Based on the satisfied contracts we can identify safety artefacts related to such contracts (e.g., test cases) that can be useful in the current context.

11.4.3 Generated Safety Arguments

Figure 11.5 shows the top level goals of the LAAP safety argument for the two systems. Both argument-fragments are generated in the same way, hence share the similar structure. To support the top-level goal that the component satisfies the allocated safety requirements, we decompose the top-level goal to argue over the following: the LAAP strong contracts are satisfied, all satisfied contracts are consistent, and the relevant weak contracts are satisfied. As all strong contracts must be satisfied in both context, the argument related to the strong contract satisfaction (Figure 11.6) is the same for both cases.

The top level goals are further decomposed to argue over satisfaction of

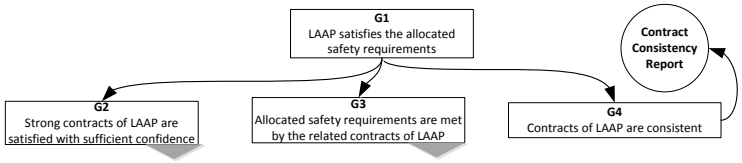

Figure 11.5: Top goals of the LAAP safety argument for both the GWL and SWL

each allocated safety requirement. As discussed in Section 11.4.2, some of the contracts are not satisfied in the GWL and in the same time some of the requirements are discarded as quality management, hence not included in the LAAP safety argument in context of GWL. In case of the GWL, SWSR2 and SWSR4 are not included in the corresponding argument (Figure 11.7), while for the SWL, all six requirements are included in the corresponding argument (Figure 11.8).

As most of the requirements are addressed by the strong contracts that are argued in a separate argument branch, the away goals are used to relate to those arguments, while the weak contracts that are used to support a requirement for the first time in the argument are further developed (e.g., the contracts *LAAP-5* and *LAAP-6* for requirements SWSR2 and SWSR5). Establishing that the safety contracts are sufficient to support a certain requirement is done by inspection.

11.5 Discussion

As described in Section 11.2.1, ISO 26262 requires certain information to be gathered during the concept phase. The standard states that software safety requirements should consider this information. In case of SEooC, this information should be assumed out-of-context and validated in-context. In the case of other reusable elements such as qualified software elements, this information should be made available and validated prior to the integration of the element into an ISO 26262 compliant system. The guidelines provided by the standard do not go into further detail but stop at the message that this information should be considered, assumed and validated. As described in Section 11.3 and demonstrated in Section 11.4, the generative reuse approach based on safety contracts provides means to assume, consider and validate this information. When developing SEooC, the required information is assumed within safety

contracts, by associating these contracts with SW safety requirements, the requirements are related and consider this information. Upon integration of a reusable component together with its safety contracts, the assumed information or information that should be made available is validated by checking that the reused safety contracts assumptions are satisfied in the particular system.

As demonstrated in Section 11.4.2, what is safety relevant in one system can sometimes be regarded as quality management in another system. This is the main reason why reusing safety artefacts (such as product-based safety argument-fragments) first needs a phase of identifying what is relevant, which is supported by the safety contracts, and after that the relevant information can be composed and the artefact reused. In the scope of our work we use the safety contracts to generate safety case argument-fragments, while there is potential to use the contracts to generate different types of safety analyses (e.g., FTA) [16] through the connection of the safety contracts with the FPTC analysis [8]. The generation of the specific safety argument-fragments is still semi-automatically performed since the integrator needs to align the assumed with the actual safety requirements. Although methods could be developed to ease the matching of the safety requirements and the associated contracts, and matching assumed and actual safety requirements, the step towards developing such methods would be formalisation of the requirements, which faces different challenges [17]. Safety contracts share some of these challenges as well. Hence we recognise the need for capturing both formal and informal aspects in the safety contracts. While the formally specified parts of the assumptions and guarantees are used for both contract-based verification and argument-fragment generation, the informal parts are only used for the arguments generation where they can be further reviewed manually.

11.6 Related Work

The ISO 26262 lack of detailed guidelines for systematic reuse has triggered researchers to align different reuse engineering methods with the standard, e.g., Product-line Engineering (PLE) and Component Based Software Engineering (CBSE). PLE can be aligned with the ISO 26262 to facilitate reuse of artefacts [11]. The proposed approach provides means to specify, manage and trace commonalities and variabilities at different parts of the ISO 26262 safety process.

Reusing safety artefacts requires that variability within them is managed. A PLE-based approach shows how variability can be integrated into the func-

tional safety models by combining functional safety and variability modelling tools [12]. Another approach focuses on Trusted Product Lines by forming a framework for demonstrating that the derived products are fit for purpose in high-integrity civil airspace systems [13]. The work aligns PLE with civil airspace safety standard recommendations on development and integration of reusable elements.

An approach that distinguishes between component types as out-of-context components and component implementations as in-context instantiations of the component types explores use of assume/guarantee contracts to facilitate reuse [14]. The work provides an incremental certification lifecycle for CBSE and outlines the role of contracts in the proposed lifecycle.

In contrast to these works we focus on providing detailed guidelines for development and integration of reusable safety elements within the safety-critical systems. Moreover, we focus on the automotive industry by aligning the provided guidelines with the ISO 26262 safety process. More specifically, we support generative reuse of safety argument-fragments since that increases the reusability of the efforts invested in capturing safety rationale within the safety contracts. We are not aware of other works with this specific focus.

11.7 Conclusion and Future Work

Safety standards, particularly ISO 26262, lack support for reuse and integration of safety elements, although modern safety-critical systems highly rely on reuse. In this paper we have presented a safety contracts development process that bases reuse of safety elements around the notion of safety contracts. We have shown on a real-world case that the safety contracts can be successfully used to complement and augment ISO 26262 safety process to provide support for reuse and integration of safety elements. Moreover, safety contracts provide a platform for generative reuse of safety artefacts by facilitating generation of safety case argument-fragments and potentially other safety analyses.

As future work we plan to develop the real-world case further and conduct series of studies to evaluate different techniques related to safety contracts. Furthermore, we intend to fully utilise the generative reuse potential of the safety contracts by looking into generation of different safety analyses. While currently only partially supported by CHESS-toolset [18], we plan to continue extension of the tool and further optimisations of the implemented contract formalism.

Acknowledgements

This work is supported by the Swedish Foundation for Strategic Research (SSF) project SYNOPSIS and the EU Artemis-funded nSafeCer project.

Table 11.4: The context statements and evidence of the LAAP safety contracts

LAAP-1_C1:	The contract is based on the specification of the Input validation and error handling of LAAP;
LAAP-1_E1	*name*: Unit testing results *description*: The evidence satisfies ASIL B requirements. *supporting argument*: -;
LAAP-2_C1:	The contract is based on the specification of the Input validation and error handling of LAAP;
LAAP-2_E1	*name*: Unit testing results *description*: The evidence satisfies ASIL B requirements. *supporting argument*: -;
LAAP-3_C1:	The contract is based on the LAAP watchdog timer configuration;
LAAP-3_E1	*name*: Watchdog inspection report *description*: The evidence satisfies ASIL A requirements. *supporting argument*: -;
LAAP-3_E2	*name*: Unit testing results *description*: The evidence satisfies ASIL B requirements. *supporting argument*: -;
LAAP-4_C1:	The contract is derived from the FPTC analysis results for the LAAP component;
LAAP-4_E1	*name*: LAAP FPTC analysis report *description*: The evidence satisfies ASIL B requirements. *supporting argument*: FPTC_analysis_conf;
LAAP-5_C1:	The contract is derived from the FPTC analysis results for the LAAP component;
LAAP-5_E1	*name*: LAAP FPTC analysis report *description*: The evidence satisfies ASIL B requirements. *supporting argument*: FPTC_analysis_conf;
LAAP-6_C1:	The contract is derived from the FPTC analysis results for the LAAP component;
LAAP-6_E1	*name*: LAAP FPTC analysis report *description*:The evidence satisfies ASIL B requirements. *supporting argument*: FPTC_analysis_conf;
LAAP-6_E2	*name*: Unit testing results *description*: The evidence satisfies ASIL B requirements. *supporting argument*: -;

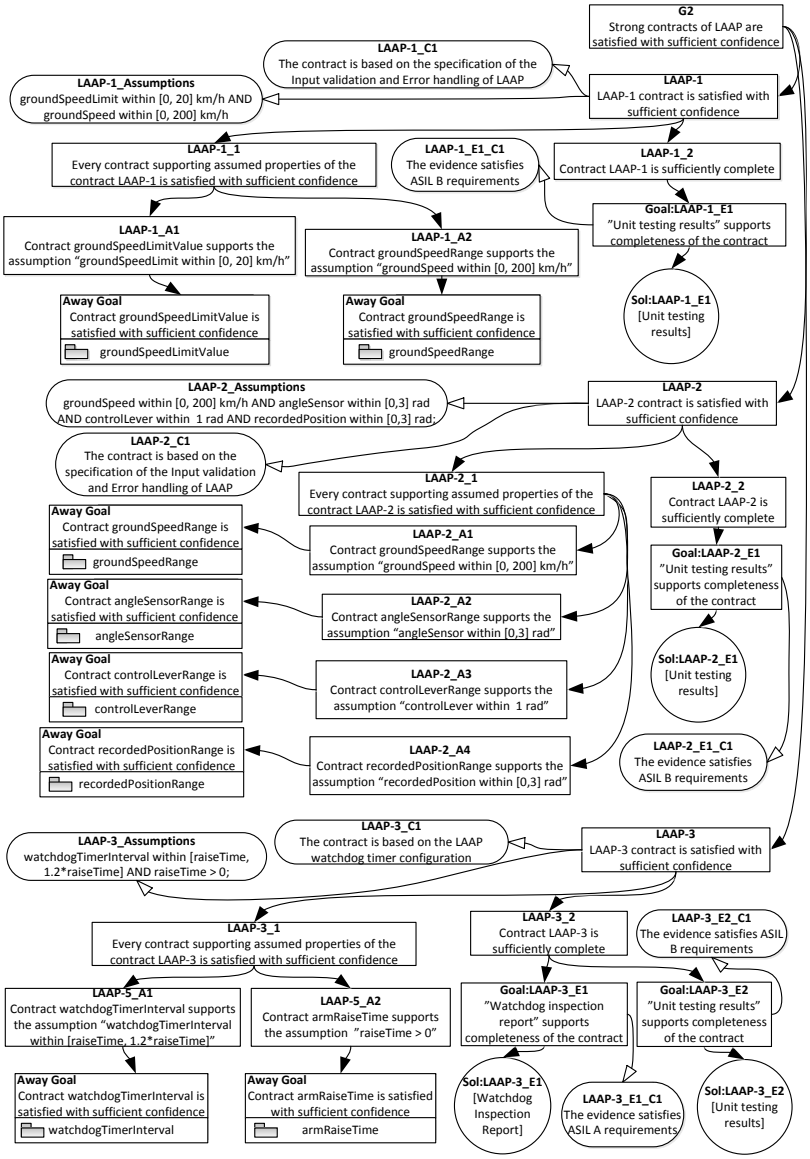

Figure 11.6: Safety argument-fragment for the strong contracts (the same for both systems)

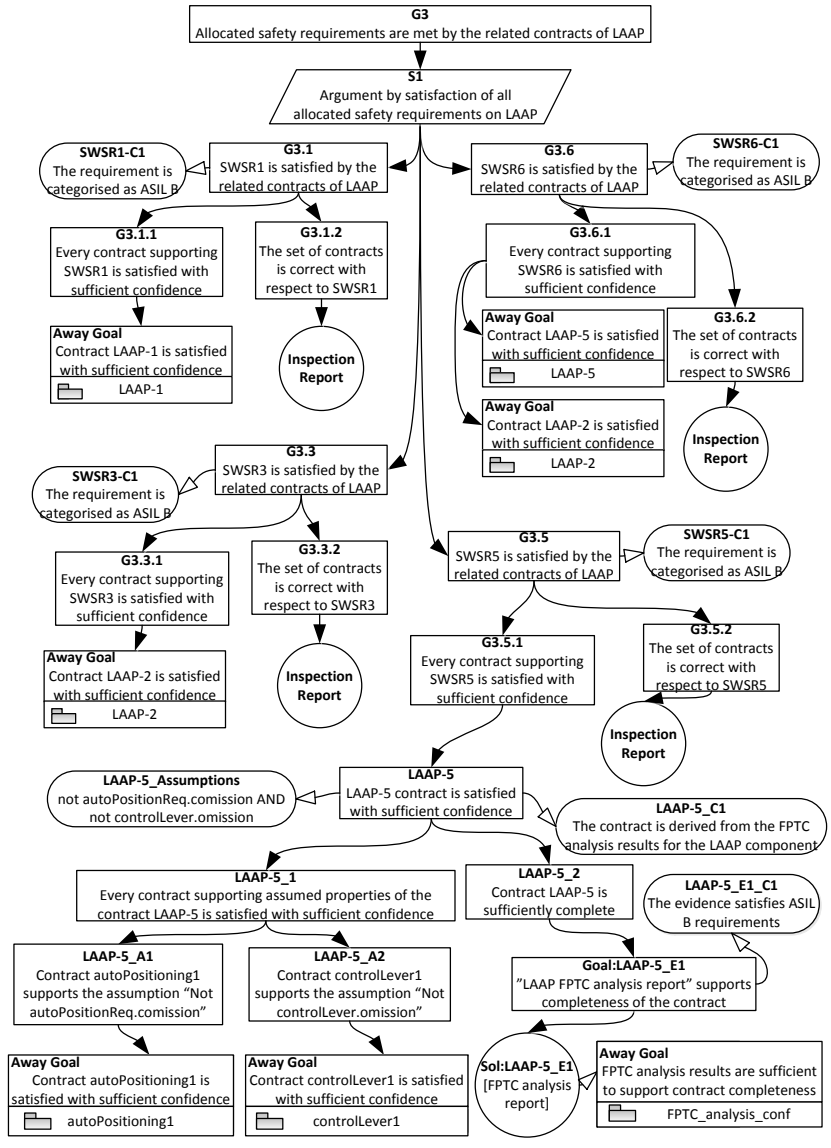

Figure 11.7: Safety argument-fragment for the safety requirements allocated on the LAAP in context of GWL

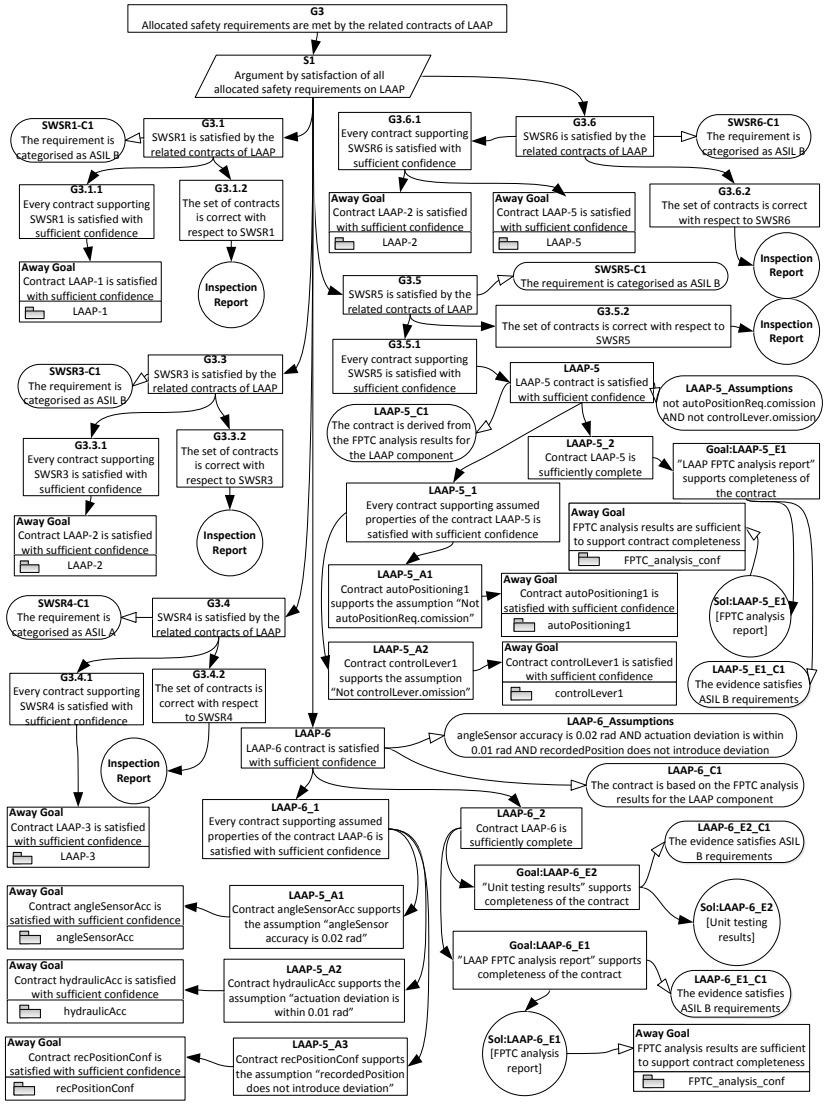

Figure 11.8: Safety argument-fragment for the safety requirements allocated on the LAAP in context of SWL

Bibliography

[1] J.-M. Astruc and N. Becker. Toward the Application of ISO 26262 for Real-life Embedded Mechatronic Systems. In *International Conference on Embedded Real Time Software and Systems*. ERTS2, 2010.

[2] S. Baumgart, J. Fröberg, and S. Punnekkat. Industrial Challenges to Achieve Functional Safety Compliance in Product Lines. In *The 40th Euromicro Conference on Software Engineering and Advanced Applications*, August 2014.

[3] International Organization for Standardization (ISO). *ISO 26262: Road vehicles — Functional safety*. ISO, 2011.

[4] B. Meyer. The next software breakthrough. *Computer*, 30(7):113–114, 1997.

[5] W. B. Frakes and K. Kang. Software Reuse Research: Status and Future. *Transactions on Software Engineering*, 31(7):529–536, 2005.

[6] Y. Papadopoulos, M. Walker, D. Parker, E. Rüde, R. Hamann, A. Uhlig, U. Grätz, and R. Lien. Engineering Failure Analysis and Design Optimisation With HiP-HOPS. *Engineering Failure Analysis*, 18(2):590–608, 2011.

[7] R. Hawkins, I. Habli, D. Kolovos, R. Paige, and T. P. Kelly. Weaving an Assurance Case from Design: A Model-Based Approach. In *16th International Symposium on High Assurance Systems Engineering*, pages 110–117. IEEE, January 2015.

[8] I. Sljivo, B. Gallina, J. Carlson, H. Hansson, and S. Puri. A Method to Generate Reusable Safety Case Fragments from Compositional Safety

Analysis. In *14th International Conference on Software Reuse*. Springer-Verlag, January 2015.

[9] I. Sljivo, B. Gallina, J. Carlson, and H. Hansson. Strong and weak contract formalism for third-party component reuse. In *International Workshop on Software Certification*. IEEE Computer Society, November 2013.

[10] I. Sljivo, B. Gallina, J. Carlson, and H. Hansson. Generation of Safety Case Argument-Fragments from Safety Contracts. In Andrea Bondavalli and Felicita Di Giandomenico, editors, *33rd International Conference on Computer Safety, Reliability, and Security*, volume 8666 of *LNCS*, pages 170–185. Springer, Heidelberg, September 2014.

[11] B. Gallina, A. Gallucci, K. Lundqvist, and M. Nyberg. VROOM & cC: a Method to Build Safety Cases for ISO 26262-compliant Product Lines. In *SAFECOMP Workshop on Next Generation of System Assurance Approaches for Safety-Critical Systems*. HAL / CNRS report, September 2013.

[12] M. Schulze, J. Mauersberger, and D. Beuche. Functional Safety and Variability: Can It Be Brought Together? In *17th International Software Product Line Conference*, pages 236–243. ACM, 2013.

[13] S. Hutchesson and J. McDermid. Trusted Product Lines. *Information & Software Technology*, 55(3):525–540, 2013.

[14] P. Graydon and I. Bate. The Nature and Content of Safety Contracts: Challenges and Suggestions for a Way Forward. In *The 20th Pacific Rim International Symposium on Dependable Computing*. IEEE, November 2014.

[15] GSN Community Standard Version 1. Technical report, Origin Consulting (York) Limited, November 2011.

[16] B. Gallina, M. A. Javed, F. U. Muram, and S. Punnekkat. Model-driven Dependability Analysis Method for Component-based Architectures. In *Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, 2012.

[17] P. Filipovikj, M. Nyberg, and G. Rodriguez-Navas. Reassessing the Pattern-Based Approach for Formalizing Requirements in the Automotive Domain. In *22nd International Requirements Engineering Conference*. IEEE, August 2014.

[18] CHESS-toolset, http://www.chess-project.org/page/download.