Mälardalen University Doctoral Dissertation 268

Assurance Aware Contract-based Design for Safety-Critical Systems

Irfan Sljivo

MÄLARDALEN UNIVERSITY
SWEDEN

Mälardalen University Press Dissertations
No. 268

ASSURANCE AWARE CONTRACT-BASED DESIGN FOR SAFETY-CRITICAL SYSTEMS

Irfan Sljivo

2018

School of Innovation, Design and Engineering

ISBN 978-91-7485-401-5
ISSN 1651-4238
Printed by E-Print AB, Stockholm, Sweden

Mälardalen University Press Dissertations
No. 268

ASSURANCE AWARE CONTRACT-BASED DESIGN FOR SAFETY-CRITICAL SYSTEMS

Irfan Sljivo

Akademisk avhandling

som för avläggande av teknologie doktorsexamen i datavetenskap vid
Akademin för innovation, design och teknik kommer att offentligen försvaras
tisdagen den 2 oktober 2018, 13.30 i Gamma, Mälardalens högskola, Västerås.

Fakultetsopponent: Associate Professor Mario Trapp,
Fraunhofer Institute for Experimental Software Engineering

Akademin för innovation, design och teknik

Abstract

Safety-critical systems are those systems whose malfunctioning can result in harm or loss of human life, or damage to property or the environment. Such 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 safety-critical systems to comply with safety standards is a time-consuming and costly process. It can often be the case that the development of the safety case is more costly than the development of the system itself.

Component-based development is a method that separates the development of the components of a system from the development of the system itself. The latter is done by composing reusable components that are developed independently of the system. Safety-critical systems require that the safety case of such components is integrated in the overall safety case of the system. For this purpose, the reusable components, together with their safety case, can be described via specifications called contracts. By checking the contracts of each component of the system against each other, it is possible to determine if the components can be composed together and still fulfil the contract specifications. Contract-based design combined with component-based development has the potential to reduce the cost and time needed to develop both the system and the accompanying safety case. Such contract-based design can then be used to facilitate reuse of parts of the system as well as verifying that the system fulfils certain requirements. While contract-based design can be used to verify that a system meets certain requirements based on its contract-specification, actually assuring that the system behaves according to the verification results require additional evidence. Hence, reuse of safety-relevant components via contract-based design is not sufficient without the reuse of the accompanying safety case artefacts, which include both the safety argument and the supporting evidence.

In this thesis we focus on developing the notion of safety contracts that can be used to make a contract-based design aware of the needs of safety assurance. The goals of such assurance aware contract-based design are to promote reuse of the assurance-related artefacts such as arguments and evidence, as well as to automate creation of parts of the safety assurance case. To address this, we explore the following research goals in more detail: (1) to facilitate automated contract-driven assurance, (2) to facilitate reuse of safety-relevant components and their accompanying assurance-relevant artefacts, and (3) to align such assurance-aware contract-based design with existing failure logic analysis. To meet the first goal, we identify the additional information needed for contract-based assurance and structure it in form of argumentation patterns of reusable reasoning. Then, we define a meta-model to connect the system modelling elements related to the contracts with the safety case elements, such as evidence and arguments. Based on this meta-model, we define an algorithm for automated instantiation of the proposed argumentation patterns from system models compliant with the proposed meta-model. To facilitate reuse of the assurance-related artefacts (goal (2)), we define variability on the contract level to distinguish between contracts that are relevant for all systems and those that are system-specific. Furthermore, we align the assurance-aware contract-based design with the ISO 26262 automotive safety standard and its reuse concepts. Finally, in addressing the third goal, we connect the assurance-aware contract-based design with an existing failure logic analysis and show how such combination can be used to automate instantiation of existing argumentation patterns. In a number of real-world examples we demonstrate and evaluate the feasibility of our contributions.

ISBN 978-91-7485-401-5
ISSN 1651-4238

Abstract

Safety-critical systems are those systems whose malfunctioning can result in harm or loss of human life, or damage to property or the environment. Such 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 safety-critical systems to comply with safety standards is a time-consuming and costly process. It can often be the case that the development of the safety case is more costly than the development of the system itself.

Component-based development is a method that separates the development of the components of a system from the development of the system itself. The latter is done by composing reusable components that are developed independently of the system. Safety-critical systems require that the safety case of such components is integrated in the overall safety case of the system. For this purpose, the reusable components, together with their safety case, can be described via specifications called contracts. By checking the contracts of each component of the system against each other, it is possible to determine if the components can be composed together and still fulfil the contract specifications. Contract-based design combined with component-based development has the potential to reduce the cost and time needed to develop both the system and the accompanying safety case. Such contract-based design can then be used to facilitate reuse of parts of the system as well as verifying that the system fulfils certain requirements. While contract-based design can be used to verify that a system meets certain requirements based on its contract-specification, actually assuring that the system behaves according to the verification results require additional evidence. Hence, reuse of safety-relevant components via contract-based design is not sufficient without the reuse of the accompanying safety case artefacts, which include both the safety argument and the supporting evidence.

In this thesis we focus on developing the notion of *safety contracts* that can be used to make a contract-based design aware of the needs of safety assurance. The goals of such assurance aware contract-based design are to promote reuse of the assurance-related artefacts such as arguments and evidence, as well as to automate creation of parts of the safety assurance case. To address this, we explore the following research goals in more detail: (1) to facilitate automated contract-driven assurance, (2) to facilitate reuse of safety-relevant components and their accompanying assurance-relevant artefacts, and (3) to align such assurance-aware contract-based design with existing failure logic analysis. To meet the first goal, we identify the additional information needed for contract-based assurance and structure it in form of argumentation patterns of reusable reasoning. Then, we define a meta-model to connect the system modelling elements related to the contracts with the safety case elements, such as evidence and arguments. Based on this meta-model, we define an algorithm for automated instantiation of the proposed argumentation patterns from system models compliant with the proposed meta-model. To facilitate reuse of the assurance-related artefacts (goal (2)), we define variability on the contract level to distinguish between contracts that are relevant for all systems and those that are system-specific. Furthermore, we align the assurance-aware contract-based design with the ISO 26262 automotive safety standard and its reuse concepts. Finally, in addressing the third goal, we connect the assurance-aware contract-based design with an existing failure logic analysis and show how such combination can be used to automate instantiation of existing argumentation patterns. In a number of real-world examples we demonstrate and evaluate the feasibility of our contributions.

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 utvecklas enligt en branschspecifik säkerhetsstandard som ofta innefattar 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. Arbetet med säkerhetsbevisningen är ofta den dominerande kostnaden i utvecklingsarbetet.

Komponentbaserad utveckling är en metod där man separerar utvecklingen av systemets komponenter från utvecklingen av systemet. Det senare utförs genom sammansättning av återanvändbara komponenter som utvecklats oberoende av det sammansatta systemet. För säkerhetskritiska system behöver dessutom komponenternas säkerhetsbevisning kombineras till en säkerhetsbevisning för det sammansatta systemet. För detta ändamål kan komponenterna, inklusive säkerhetsbevisningen, beskrivas av specifikationer som kallas kontrakt. Genom att kontrollera kontrakten för varje komponent i systemet mot varandra är det möjligt att bestämma om komponenterna kan integreras och fortfarande uppfylla sina kontraktsspecifikationer. Kontraktbaserad design kombinerad med komponentbaserad utveckling har potential att minska kostnaden och tiden som behövs för att utveckla både systemet och den medföljande säkerhetsbevisningen. Sådan kontraktsbaserad design kan sedan användas för att återanvända delar av systemet samt att verifiera att systemet uppfyller vissa krav.

Även om kontraktsbaserad design kan användas för att verifiera att ett system uppfyller vissa krav baserat på dess kontraktsspecifikation, behövs det fortfarande ytterligare bevis för att garantera att systemet uppför sig i enlighet med de krav på säkerhetsbevisningen som behöver uppfyllas. Därför är återanvändning av säkerhetsrelaterade komponenter via kontraktsbaserad design

inte tillräcklig utan att samtidigt återanvända de medföljande delarna av säkerhetsbevisningen som innehåller säkerhetsargument och stödjande bevis.

I den hr avhandlingen presenterar vi en form av säkerhetskontrakt som kan användas för att utöka kontraktbaserad design till att även inkludera säkerhetsbevisning. Målen med en sådan kontraktsbaserad design är att främja återanvändning av delar av säkerhetsbevisningen, men också att automatisera skapandet av densamma. I avhandlingen undersöker vi följande i detalj: (1) automatiserad generering av kontraktbaserad säkerhetsbevisning, (2) återanvändning av säkerhetsrelevanta komponenter och deras medföljande säkerhetsbevisningselement och (3) anpassning av sådan kontraktbaserad design till befintliga felanalyser. För att närma oss (1), identifierar vi den ytterligare information som behövs för kontraktsbaserad argumentation och strukturerar den i form av argumentationsmönster. Därefter definierar vi en metamodell för att koppla ihop systemmodelleringselementen till kontrakt med säkerhetsbevisningselement. Baserat på denna metamodell definierar vi en algoritm för automatiserad instansering av de föreslagna argumentationsmönstren från systemmodeller som överensstämmer med den föreslagna metamodellen. För att underlätta återanvändning av delar av en säkerhetsbevisning (2 ovan) definierar vi variabilitet på kontraktsnivån för att skilja mellan kontrakt som är relevanta för alla system och de som är systemspecifika. Dessutom anpassar vi sådan kontraktsbaserade design med säkerhetsstandarden ISO 26262 för vägfordon och dess återanvändningskoncept. Slutligen adresserar vi (3) genom att ansluta den kontraktsbaserade designen till en befintlig felanalys och visa hur en sådan kombination kan användas för att automatisera instansiering av befintliga argumentationsmönster. Vi använder genomgående ett verkligt fall för att demonstrera och utvärdera genomförbarheten av de föreslagna bidragen.

In loving memory of my mother

Acknowledgments

There are many to whom I owe my gratitude for supporting me in taking this PhD path in the first place, and perhaps more importantly, supporting me throughout the years on this path. First and foremost, I would like to thank my family for their endless love, inspiration and support they have given me. In particular, I would like to thank my parents for being the source of guidance and inspiration that made this journey manageable. Sadly, my mother did not live to hold this thesis in her hand, which is as much hers as much it is my success. I dedicate this thesis to her and my father, for there is a great deal of them in it as well.

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.

My deepest gratitude goes to my co-authors as well as the members of SYNOPSIS, SAFECER, SAFECOP, AMASS and FIC research projects for all the positive influence they had on my research. I am extremely grateful to Hans Hansson, Barbara Gallina, Jan Carlson, Patrick Graydon, Iain Bate, Sasikumar Punnekkat, Ibrahim Habli, Bernhard Kaiser and Henrik Thane for all the useful discussions and the vast knowledge they have shared with me. An enormous thank you goes to Omar Jaradat with whom I have worked the most throughout the years. Thank you for all the discussions, for putting up with me during our conference and meeting trips, and for being not just a colleague, but a friend and a brother.

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 Nehren-

heim and Harold Lawson. I would also like to thank the IDT administration staff for their support with practical issues. Special thanks goes to Carola, for without her many of the administrative tasks would be a nightmare.

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. I will not enumerate each and every one of you, because I am sure I would miss someone very dear, as I am very grateful to have met you all. However, there are a few persons that I simply must mention: Aida, Adnan, Svetlana, Elena, Momo, Sara, Saad and Leo. You have been there for me all the way from the beginning. I am very grateful to have you as my friends. A special thanks goes to my office mates throughout the years Omar, Gabriel, Anita, Husni, Filip and Julieth for without them the light in our office would be rarely on.

The work in this thesis has been supported by the Swedish Foundation for Strategic Research (SSF) via the projects SYNOPSIS[1] and FIC[2] as well as EU and VINNOVA via the Artemis JTI project SafeCer[3] and ECSEL Joint Undertaking projects AMASS[4] (No 692474) and SAFECOP[5] (No 692529).

Irfan Šljivo
September, 2018
Västerås, Sweden

[1] http://www.es.mdh.se/SYNOPSIS/
[2] http://www.es.mdh.se/fic
[3] https://www.es.mdh.se/projects/294-
[4] https://www.amass-ecsel.eu/
[5] http://www.safecop.eu/

List of publications (short)[6]

The following is the list of publications that the thesis is mainly based on:

Paper A *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 B *Using Safety Contracts to Guide the Integration of Reusable Safety Elements within ISO 26262*, Irfan Šljivo, Barbara Gallina, Jan Carlson, Hans Hansson. In Proceedings of the 21st IEEE Pacific Rim International Symposium on Dependable Computing (PRDC), IEEE, November 2015.

Paper C *A Method to Generate Reusable Safety Case Fragments from Compositional Safety Analysis*, Irfan Šljivo, Barbara Gallina, Jan Carlson, Hans Hansson, Stefano Puri. Journal of Systems and Software: Special Issue on Software Reuse 131, C (September 2017), 570-590. DOI: https://doi.org/10.1016/j.jss.2016.07.034.

Paper D *Tool-Supported Safety-Relevant Component Reuse: From Specification to Argumentation*, Irfan Šljivo, Barbara Gallina, Jan Carlson, Hans Hansson, Stefano Puri. In Proceedings of the 23rd International Conference on Reliable Software Technologies (Ada-Europe), June 2018.

Other relevant publications this thesis builds upon (in chronological order):

1. *Fostering Reuse within Safety-critical Component-based Systems through Fine-grained Contracts*, Irfan Šljivo, Jan Carlson, Barbara Gallina, Hans

[6]The full list of papers is presented in Appendix A

Hansson. International Workshop on Critical Software Component Reusability and Certification across Domains (CSC2013), Jun 2013.

2. *Strong and Weak Contract Formalism for Third-Party Component Reuse*, Irfan Šljivo, Barbara Gallina, Jan Carlson, Hans Hansson. In Proceedings of the IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 3rd International Workshop on Software Certification (WoSoCer 2013), November 2013.

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

4. *A Method to Generate Reusable Safety Case Fragments from Compositional Safety Analysis*, Irfan Sljivo, Barbara Gallina, Jan Carlson, Hans Hansson, Stefano Puri. In Proceedings of the 14th International Conference on Software Reuse (ICSR2015), January 2015.

5. *Deriving Safety Contracts to Support Architecture Design of Safety Critical Systems*, Irfan Šljivo, Omar Jaradat, Iain Bate, Patrick Graydon. In Proceedings of the16th IEEE International Symposium on High Assurance Systems Engineering (HASE 2015), January 2015.

6. *Using Safety Contracts to Guide the Integration of Reusable Safety Elements within ISO 26262*, Irfan Šljivo, Barbara Gallina, Jan Carlson, Hans Hansson. MRTC Report, Malardalen Real-Time Research Centre (MRTC 2015), March 2015.

7. *Facilitating Reuse of Safety Case Artefacts Using Safety Contracts*, Irfan Šljivo. Licentiate Thesis. Mälardalen University Press. June 2015.

8. *Configuration-aware Contracts*. Irfan Šljivo, Barbara Gallina, Jan Carlson, Hans Hansson. In Proceedings of the 4th International Workshop on Assurance Cases for Software-intensive Systems (ASSURE2016), September 2016.

9. *Assuring Degradation Cascades of Car Platoons via Contracts*, Irfan Šljivo, Barbara Gallina, Bernhard Kaiser. In Proceedings of the 6th International Workshop on Next Generation of System Assurance Approaches for Safety-Critical Systems (SASSUR-2017), September 2017.

Contents

Chapter 1

Introduction

Safety-critical systems are those systems whose unintended or malfunctioning behaviour 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 between 70 and 100 embedded computers on board, with overall software that scales up to 100 million lines of code[1]. These safety-critical software-intensive systems need to meet certain requirements to achieve sufficient levels of safety. Compliance with a set of domain-specific safety standards is one of the requirements. 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 line of code [3].

In most cases, safety standards require a safety case to assure that any unacceptable 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 the artefacts (e.g., results of failure analyses or verification evidence) produced during the certification process, 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).

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

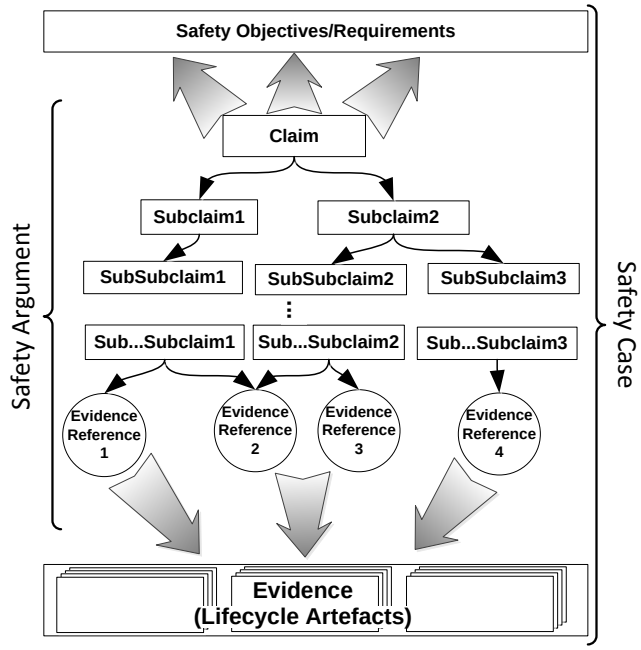

Figure 1.1: The role of safety argumentation within a safety case (adapted from [4])

More and more safety standards are offering support for reuse to reduce the production costs and time needed to achieve certification. For example, in the automotive functional safety standard (ISO 26262) [5] reuse is explicitly supported through the notion of Safety Elements out of Context (SEooC). Whether reuse is planned (systematic) or "ad hoc" (non-systematic) has significant influence on the safety of the system [6]. Hence, safety standards typically take into consideration whether the safety element being reused is developed for reuse or not. For example, the SEooC concept is used for elements that are developed for reuse and according to the standard, such as a real-time operating system or a computer vision system. The planned reuse assumes that the elements being reused have been developed with reuse in mind, which usually results in higher development cost for the reusable element, but the return on this investment is obtained if the element is reused often enough [7]. 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.

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 [8]. 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 [9]. Besides the support for reuse and independent development, contract-based design in CBSE enables contract-based compositional verification of properties on a system model such that its results can be used as evidence in assuring that the system is acceptably safe. A contract is a pair of assertions, namely assumptions and guarantees, such that the component offers the guarantees about its own behaviour, given that the environment in which it is deployed fulfils the assumptions [10].

1.1 Problem Statement and Research Goals

Industries developing safety-critical systems often have problems with high production costs and the time needed to achieve safety certification for software-intensive systems. One way of addressing this issue is by automating parts of the assurance efforts and enabling reuse of not only software components, but also the accompanying certification-relevant artefacts. To this end, contract-based design within CBSE has supported reuse of components and compositional verification as a way to partially assure certain properties of the components. Using such contract-based design in safety-critical systems to cut down the cost of certification rises two challenges:

- while compositional verification of a system using contracts establishes validity of a particular property of the system model, additional assurance is required to build the confidence that the system implementation actually exhibits that property, and
- contract-based design intrinsically supports reuse of components for environments defined by the contract assumptions, but lacks support for accompanying assurance information reuse as well as reusable safety-relevant components, such as SEooC, which are characterised by different behaviours in different environments.

Contributing to the assurance- and reuse aware contract-based design, aiming to cut down the cost of certification, drives our research and leads to the overall goal of this thesis:

Overall Goal: *To facilitate automation of assurance and fine-grained reuse of assurance information by contract-based design.*

To move the state-of-the-art towards the overall goal, we decompose it into three subgoals that we focus on in the thesis:

- **Research Goal 1:** *To facilitate automated contract-driven assurance in order to reduce the overall assurance efforts.*

  Since safety assurance is driven by the safety requirements allocated to the different components of the system, the corresponding contracts of those components are envisaged to realise the allocated requirements. In such a scenario, each requirement is realised by a set of contracts and its validity is supported by contract-based compositional verification. But since the results of compositional verification are not sufficient for assuring that the system fulfils the requirement, the additional information needed for assuring that a system satisfies the given safety requirement should be identified and structured according to an established argumentation strategy. If such a strategy is captured in form of argumentation-patterns, those patterns can be automatically instantiated from the system models only when the relevant assurance information is clearly connected with the system modelling artefacts. In order to meet this goal, we first identify the additional assurance information needed for contract-driven assurance and capture them in form of argument patterns, we define how this information should be structured within assurance-aware system models, and detail how the argument patterns can be automatically instantiated from the enriched system models.

- **Research Goal 2:** *To facilitate reuse of SEooC and their context-specific assurance artefacts by contract-based design.*

  Reusable components such as SEooC are characterised by certain parameters that tailor their behaviour for different contexts. While the support

for reuse in contract-based design has been mainly focused on components (i.e., implementations of contracts), it has not focused on reusable components as implementations of a set of contracts for different environments that may or may not be satisfiable together. This context variability on the contract level is needed to identify which properties offered by the reusable component are relevant in the system in which the component is currently reused. Hence, the assurance information related to those properties captured in the contracts can also be identified as relevant for reuse. In order to meet the second research goal, we explore how to achieve the context variability on the contract level and in which way it can be adopted in the existing SEooC development and integration process.

- **Research Goal 3:** *To support reuse of results from existing failure logic analyses and automation of assurance based on those results through the assurance- and reuse aware contract-based design.*

  Just as hazard analysis is the basis for safety engineering at the system level, derivation of contracts and identification of the corresponding assumptions plays a similar role at the component level [11]. While many works deal with contract formalisms and what those contracts should be used for, the lack of clear guidelines and methods for their derivation hinders their adoption in existing safety processes. In order to meet the third research goal, we explore if and in which way methods for contract derivation from existing failure logic analyses can be used to reuse the results of failure logic analyses and automate parts of the assurance based on those results.

1.2 Contributions

Addressing the three research goals yielded the main contributions of this thesis.
In meeting research goal 1, we identify the following contributions:

- **The introduction of argumentation patterns to capture the contract-driven assurance reasoning**: We identify the additional assurance information needed to assure with sufficient confidence that a system behaves according to the results of the compositional contract-based veri-

fication of its model. We structure the information in form of contract-driven assurance argumentation patterns.

- **Connecting the contract-based system modelling and assurance case modelling on the meta-model level**: We structure the system and assurance information needed for contract-driven assurance in form of a Safety Element Meta-Model (SEMM).

- **A method for automated instantiation of the contract-driven assurance argumentation patterns from system models compliant with SEMM**: We map the SEMM elements to the assurance elements and present a set of rules for automated instantiation of contract-driven assurance argument patterns.

In meeting research goal 2, we identify the following contributions:

- **The introduction of strong and weak contracts to manage context variability at the contract level**: We use strong and weak contracts as a way to distinguish between the properties that must be met in every environment in which a component is reused (the strong contracts), and the properties that are environment specific and do not have to be met in every environment (the weak contracts).

- **An extension of SEMM to support context variability across contexts**: We introduce the strong and weak contract modelling support in SEMM to provide greater support for out-of-context modelling, making it a Safety Element *out-of-Context* Meta-Model (SEooCMM). The extension with strong and weak contracts allows for modelling the variability of the component properties, and in that way also for modelling the variability of the associated assurance information.

- **Alignment of SEooC development with contract-based design assurance and reuse**: We present a safety contract development process and align it with the safety process recommended by ISO 26262 to concretise the systematic reuse of SEooC using assurance and reuse-aware contract-based design. The process as such is however more general, and could be aligned also with processes of similar concepts in other domains such as the avionics concept of Reusable Software Components presented in Advisory Circular 20-148 [12].

In meeting research goal 3, we identify the following contributions:

- **A method for contract derivation from compositional failure logic analysis**: We present a method for deriving safety contracts from Failure Propagation and Transformation Calculus (FPTC) analysis [13] that allows for calculation of system level behaviour from the behaviour of the individual components established in isolation.

- **An approach for instantiation of FLA-based argumentation patterns from the derived contracts**: We use the fault propagation contracts derived from the FPTC analysis to instantiate the existing argument pattern regarding the absence of Hazardous Software Failure Modes.

All technical contributions are evaluated through industrial case studies. We integrate the technical contributions in two groups and evaluate each group in a separate case study:

- **Case study 1**: We evaluate the contributions related to the first two research goals in this case study. We apply the contract-aware SEooC development and reuse to evaluate the feasibility of fine-grained reuse of a component and its assurance information, including the reuse of automatically instantiated safety assurance argument patterns.

- **Case study 2**: The main focus of this case study is to evaluate the third research goal contributions using the reuse and assurance aware contract-based design shaped by the contributions to the first two research goals. In particular, we evaluate the feasibility of reuse of results obtained from Fault Propagation and Transformation Calculus (FPTC) analysis and safety argumentation that builds upon such results.

1.3 Research Methodology

The goal of the research conducted in this thesis is to construct new methods, techniques and theoretical foundations based on 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* [14]. 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 [15]. 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.

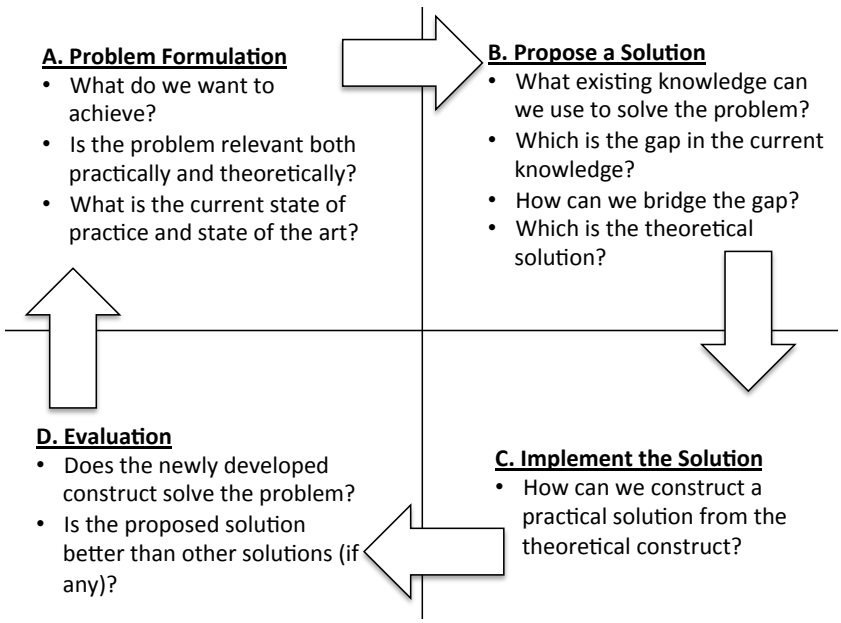

Figure 1.2: The cycle of the research process (adapted from [16])

The generic cycle of research within the majority of engineering sciences can be described in four high-level steps [16]. Figure 1.2 presents an overview of our adaptation of these 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 [15] deals with both theoretical and practical problems, our research process consists of two nested instances of the generic research cycle, one cycle for the theoretical and one for the practical/engineering problem (Figure 1.3). 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 pro-

duction usually happens [14]. 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 goals. 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, the practical cycle of the process continues to integrate the solutions to the research problem into a practical solution for 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.

In our particular case we have relied on cooperation with our industrial partners to identify those practically relevant real-world problems, which are at the same time not addressed by the state-of-the-art. Since the initially identified problem in cooperation with industry (e.g., the cost of certification) is quite broad and generic, we simplify it to a set of sub-problems that are practically feasible to evaluate. Then, we define a set of research goals to address those sub-problems, which should in turn lead us towards the solution of the generic problem. During the research of each of the goals we use simpler evaluation examples, appropriate to quickly iterate the evaluation of the evolving solution of the research goal. Once we reach a satisfactory solution, we publish it in a workshop or a conference paper. After answering a set of research goals, we integrate the solutions and evaluate the integration on a more complex, real-world examples. We aim at journal publications with such integrated contributions.

Software engineering research often relies on the 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 [17]. Exploratory case studies are usually conducted prior to the constructive work to gain deeper understanding of the research problem and the domain [14]. Upon development of new constructs, case studies can be used to evaluate a newly developed method or technique in its real-world setting in form of an explanatory case study. In some iterations of our research cycle we use case studies to evaluate the newly proposed methods.

Considering that the problem we are addressing comes from the industrial needs of our partners, we rely on real-world examples to evaluate our contributions. In our work we use both explanatory and exploratory case studies.

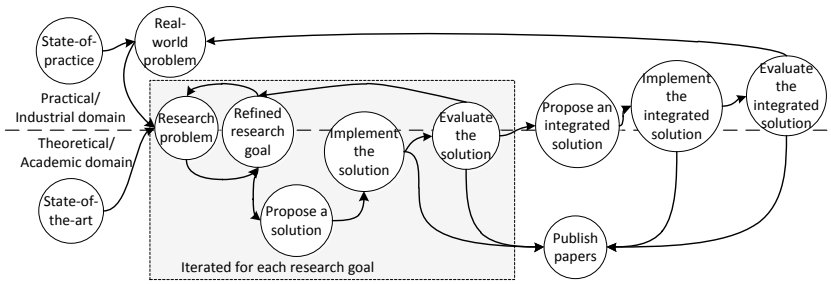

Figure 1.3: Overview of our research process

During the iterative process of answering a single research goal, we often start by performing exploratory case studies on simpler examples, and then move towards more complex real-world examples for explanatory purposes. In particular, the two case studies included in this thesis represent the explanatory case studies of the integrated solutions on complex real-world examples.

1.4 Thesis outline

The thesis is organised as a monograph based on already published research papers. The different chapters of the thesis are influenced by the different research papers and all the contributions have already been peer-reviewed as contributions of the published papers. The short list of the relevant papers is available at the beginning of the thesis, while the extended paper list together with their abstracts and remarks is available in Appendix A.

The rest of the thesis is organised as follows: In Chapter 2 we present general background information. In particular, we provide an introduction to safety assurance, safety standards and their reuse support, the most common software reuse technologies, and contract-based design. In Chapter 3, we present the real world examples that we use to demonstrate and evaluate the thesis contributions. We present our contract-driven assurance methodology in Chapter 4, while Chapter 5 presents its support for reuse. In Chapter 6 we present the method for reuse of compositional failure logic analysis results using the assurance and reuse aware contract-based design. We present the tool implementation of the proposed extensions to contract-based design in Chapter 7. In Chapter 8 we present the two industrial case studies we use for evaluation of the proposed contributions. The case studies are based on the real-world

examples introduced in Chapter 3. We present related work in Chapter 9 and finally, we bring conclusions and outline future work in Chapter 10.

Chapter 2

General Background

In this chapter we first present an overview of safety-critical systems, their development and assurance. We introduce the automotive functional safety standard ISO 26262 and the concepts from the standard that we use in this thesis. Furthermore, we present an overview of the most common software reuse technologies. Finally, we provide an introduction to the notion of contracts and detail the tool supported assumption/guarantee contract framework that we have built upon when providing the tool-support for contract-driven assurance and reuse.

2.1 Safety Assurance

In this section we introduce the essential safety terminology and focus on the assurance of safety-critical systems. In particular, we present the overview of different techniques for documenting the assurance efforts.

2.1.1 Safety Terminology

Safety is usually defined as *"freedom from unacceptable risk"* [18], where **risk** is a *"combination of the probability of occurrence of harm and the severity of that harm"* [18]. 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" [19].

When dealing with risk, we distinguish between tolerable and residual risks. **Tolerable risk** is defined as *"risk which is accepted in a given context based on the current values of society"* [18]. While the **residual risk** is defined as *"risk remaining after protective measures have been taken"* [18]. 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", but this definition is too generic, as almost any system state can be a 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"* [20].

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"* [18]. Hence there is a clear distinction between hazards and failures, since failures can occur without causing a hazard. This distinction reflects itself in the potential consequence of a hazard and a failure. While a hazard leads to harm, an occurrence of a failure will not necessarily lead to harm. This is the crucial difference between safety and reliability [21], while safety deals with hazards, **reliability** deals with failures and is defined as *"continuity of correct service"* [22].

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* [22]. 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* [22]. A cause of an error is called a **fault**. A fault is *an event that manifest itself in form of an error* [22]. 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** [22].

As mentioned in Chapter 1, a safety case is composed of all the work prod-

ucts 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 [23]. 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 its assurance requirements."* [24]. A **claim** is defined as *"a proposition being asserted by the author or utterer that is a true or false statement"* [24], 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"* [24].

2.1.2 Assurance Case Representation

The safety assurance case argument can be represented in different ways ranging from free text to more formal notations [25]. 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 typically summarised in natural language, although some use tabular structures to capture the rationale in a structured form, and others have started using graphical notations.

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., implementation 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., related to the correctness of a proof tool) 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. Two most popular graphical notations for representing the safety case arguments are: Goal Structuring Notation (GSN) [4]; and Claims, Arguments and Evidence (CAE) [26]. 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) [24] 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 this section we provide basic information about GSN.

Goal Structuring Notation

The Goal Structuring Notation (GSN) [27] 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 2.1. 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 and solutions, while the **inContextOf** relationship is used to connect the goals and strategies with supporting elements such as contexts, justifications and assumptions.

Many arguments are developed using similar rationale for structuring the

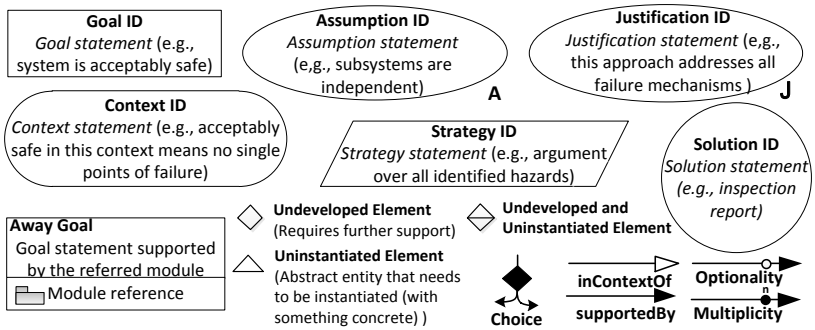

Figure 2.1: Overview of the GSN elements

goals and their supporting sub-goals. Reusable argument patterns are created as a way to capture that rationale by generalising the details of a specific argument. The main functionality of GSN has been extended to support representation of patterns of reusable reasoning [28]. To represent such argument patterns, GSN has been extended to support structural and entity abstraction [27]. The bottom row in Figure 2.1 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 [27]. Instead of further developing a certain goal, if that goal is already developed in another part of the argument, an away goal can be used to point to the original element where the goal has been developed.

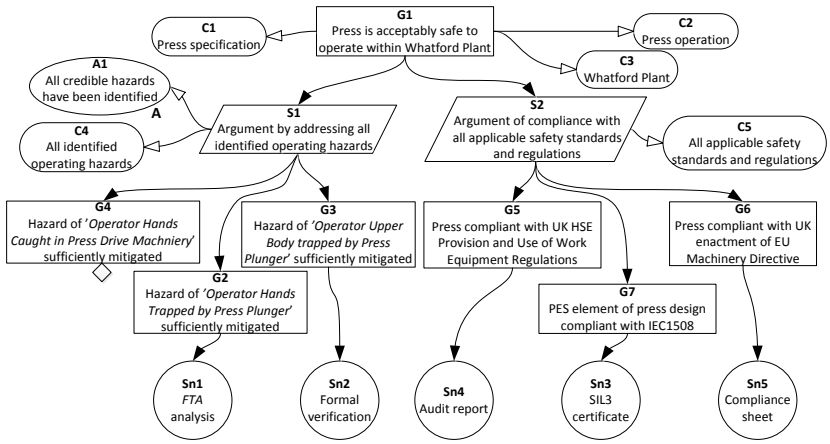

Figure 2.2: An argument example represented using GSN (adapted from [27])

An example of the application of the core of GSN is shown in Figure 2.2. 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.

2.2 Brief Overview of the Relevant Safety Standards

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 this section we briefly present some of the standards that have influenced the research presented in this thesis.

2.2.1 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 2.3. The presented lifecycle does not explicitly include verification activities, but they are required after each phase of the system development.

2.2.2 Railways Industry Standards: CENELEC EN 5012x

This group of standards for the railways industry represent 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

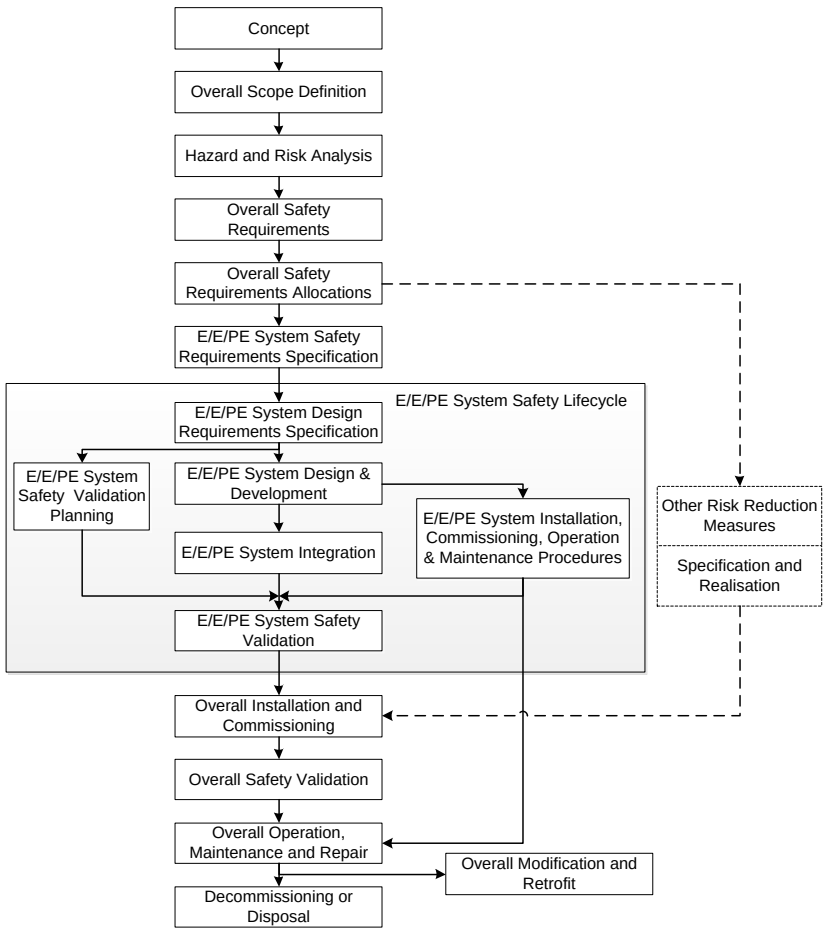

Figure 2.3: A portion of the Overall Safety Lifecycle adapted from [29]

signalling railway applications. EN 50129 explicitly requires a safety case to be provided and even defines its content.

2.2.3 Automotive Industry Standard: ISO 26262

Electronic and electrical systems (E/E) are increasingly used to implement critical functions within road vehicles. Many of the participants in traffic are exposed to safety risks due to the possible malfunctioning behaviour of those systems. 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.

2.2.4 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 [35] “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 assess-

ment methodologies, EUROCAE/SAE published a document ED-135/ARP-4761 [36] "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 [37] "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.

2.3 ISO 26262 Overview

As mentioned in Section 2.2, ISO 26262 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 [38].

Central part of Fig. 2.4 shows the safety process of the ISO 26262 standard. The process starts with the *Concept phase* (Part 3 of the standard [39]) 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). **ASIL** represents one of four safety integrity levels to specify the necessary requirements for system items imposed by ISO 26262 as well as safety measures that should be applied to avoid unreasonable risk [38]. **Safety goals** as the top-level safety requirements are established for each hazard, and further refined into **Functional Safety Requirements** (FSR), which specify implementation-independent safety behaviour. This requirement decomposition and their allocation to architectural elements makes the **functional safety concept**.

In the first part of the *Product development at system level* phase [40], the **Technical Safety Requirements** (TSR) as implementation-specific require-

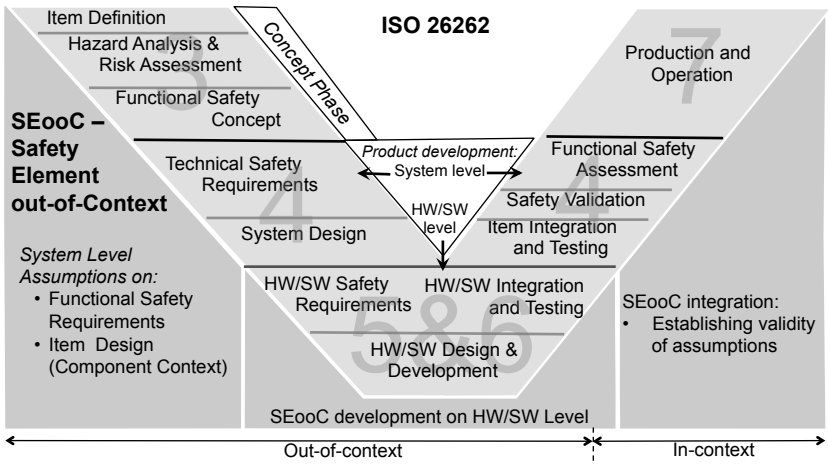

Figure 2.4: *Safety contracts development* and *SEooC development and integration* processes combined

ments, are derived from the functional safety concept. The specification and allocation of the TSR to system elements makes the **technical safety concept**. Based on the system design according to the specified technical safety concept, development and testing of both the hardware (HW) and software (SW) elements is performed. During *Product development at HW/SW level* (Parts 5 [41] and 6 [42] shown in Fig. 2.4) 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* [43] 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.

Given above, the information that needs to be gathered during concept and system design 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 and provides guidelines for the different reuse scenarios [5]: (1) elements that have been developed for reuse according to ISO 26262 in form of Safety Element out-of-Context (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. We further focus on the SEooC reuse notion, as it targets the elements developed specifically for reuse and according to the standard.

2.3.1 Safety Element out of Context

SEooC can be an element used to compose an item, but it cannot be an item since an item implements functions at vehicle level, while 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 (Figure 2.4), 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. The corresponding SW/HW requirements are derived and the component is developed to implement the requirements, after which the requirements are verified and the component is tested. To reuse SEooC in a particular system, the assumptions made during its development need to be validated in the context of that system.

2.4 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 [44]. **Software reuse** is defined as *"the use of existing software or software knowledge to construct new software"* [45]. 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" [45].

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.

2.4.1 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"* [9]. 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].

2.4.2 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* [7]. 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 [7]. 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.

2.5 Contract-based design

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 [54]. 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 [55], with **correctness** of software defined as *"the ability of software products to perform their exact tasks, as defined by their specification"* [55], and **robustness** defined as *"the ability of software systems to react appropriately to abnormal conditions"* [55].

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 [56] and Hoare [57]. 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 [58], and their application to program design by Jones [59]. Furthermore, profound influence on the Design by Contract technique and the object oriented interface design was by Parnas and his work on information hiding [60, 61], and Dijkstra with his work that coined the phrase "separation of concerns" [62].

The pre- and postcondition contracts for sequential programs were extended to support concurrent programs by using rely/guarantee rules [63]. 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. [64] 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 that 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 the 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 composition 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 [65] 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) [65]. In the reminder of the section we provide the basic notions of the particular assumption/guarantee contract theory used in this thesis [66].

2.5.1 Logic-based Contract Refinement Checking

OCRA (Othello Contracts Refinement Analysis) [66] is a tool for compositional verification of logic-based contract refinement built upon the Othello System Specification (OSS) language. The component specification with OSS includes interface and contracts specification as well as the enumeration of the subcomponents, their connections and the refinement of the contracts. The interface is composed of the ports and parameters which make the set of V variables. While ports represent input/output of the component, the parameters represent variables with constant values. The contracts are specified over the set V using the Othello constraint syntax. The subset of the syntax is shown in Table 2.1 where *constant* is a constant number and *variable* is a string. The syntax includes some temporal logic operators such as *always*.

The semantics of both components and contracts is built around the notion of a **trace**, i.e., the observable part of an execution of a component. Following the trace-based semantics, a **component** S is described with a set V_S of variables that are visible outside of the component, and a set of all traces over V_S is denoted as $Tr(V_S)$. Then, an environment of S is a subset of $Tr(V_S)$.

Table 2.1: A subset of the Othello constraints syntax

constraint :=	atom | **not** constraint | constraint **and** constraint | constraint **or** constraint | constraint **implies** constraint | **always** constraint | **never** constraint;
atom :=	**true** | **false** | term relational_op term | term;
term :=	variable | constant;
relational_op :=	("=" | "!=" | "<" | ">" | "<=" | ">=");

Assuming an assertion language, an assertion A can be described by an associated set of ports V_A and a semantics $[\![A]\!]$ defined as a subset of $Tr(V_A)$. Building on top of the assertion language, a contract C= (A, G) of the component S is a pair of assertions namely assumptions (A) and guarantees (G) over V_S. An environment E is said to be a correct environment of C iff $E \subseteq [\![A]\!]$. Contract *refinement* represents the backbone of checking the component decomposition [66]. Informally, a set of contracts of the sub-components *refines* a contract of the composite component if:

- the assumptions of all sub-component contracts are met by the other sub-components and the environment defined by the assumptions of the composite component contract; and

- the sub-component contracts deployed in the environment defined by the composite contract assumptions imply the composite contract guarantees.

Consider a component with two input ports *input1* and *input2* and two output ports *output1* and *output1Valid*, such that the output1Valid indicates the validity of the output1 signal. The example in Table 2.2 illustrates a contract for such a component using Othello constraints syntax. The contract assumes both inputs to be always positive numbers, and that the difference between the two never exceeds 0.1. In return, if these conditions are met, then the output1 is guaranteed to be always valid, i.e., the output1Valid is always true.

Table 2.2: A contract example using Othello constraints syntax

A: always (input1 $>$ 0 and input2 $>$ 0 and (input1-input2 $>=$ -0.1 and input1-input2 $<=$ 0.1));
G: always output1Valid;

2.5.2 System modelling with contract-based design support

CHESS provides an editor to model all phases of system development: from requirements definition, architecture modelling to software design and its deployment to hardware. In the CHESS toolset, components can be modelled as component types or component instances. **Component types** can be seen as elements out of context, and **component instances** as the in-context representation of the corresponding component types. Component instances inherit the attributes of the corresponding component type. System modelling in CHESS includes support for contract-based design, which relies on describing behaviours of components in terms of contracts. CHESS facilitates an interface with OCRA, such that the CHESS model together with the contracts can be exported in the Othello System Specification (OSS) format used by OCRA. The OSS syntax does not explicitly distinguish between component types and component instances. The system component is considered a component instance, while every other component is considered a component type. The component types are instantiated as subcomponents, while their variables are instantiated in the connection clause of the parent component. The contract checking with OCRA is initiated from CHESS and the result is back-propagated to the CHESS model.

Chapter 3

Real World Examples

In this chapter, we describe two selected real-world system examples that we have used to illustrate and evaluate our research. We present the system descriptions here, and use them in the reminder of the thesis as running examples, as well as for validating the research in specific case studies performed on these systems.

3.1 Fuel Level Estimation System

Fuel Level Estimation System (FLES) is based on a real estimation system used in Scania trucks with liquid fuel. The models used do not cover all the details of the real system. The main functionality of FLES is to estimate the fuel level in the truck tank and present it to the driver via dashboard. The system must be able to handle the dynamics of a driving vehicle in various driving environments, e.g., hills and off-road conditions.

3.1.1 The FLES Architecture

The architecture of FLES is shown in Figure 3.1. 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 variations 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

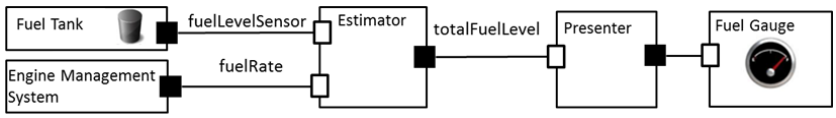

Figure 3.1: Fuel Level Estimation System High-Level Architecture

the transformations to the physical properties of sensors and its environment (e.g., size of the tank), these parameters are made configurable to make the *Estimator* usable in different variants of the system (e.g., trucks with different tank sizes).

3.1.2 The Criticality of the System

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 being informed, which would cause an unexpected engine stop. If this happens while driving on e.g., a highway, the consequences could be catastrophic. Although there may be other hazards in the system, we consider only *H1: running out of fuel while driving*.

The safety analysis, as recommended by the relevant safety standards, 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 safety goal and derived functional safety requirement:

- *SG1*: The FLES shall not show higher fuel level on the fuel gauge than the actual fuel in the vehicle's tank.
- *FSR1*: The Estimator component shall not provide value of the estimated fuel level that deviates more than -5% from the actual fuel-level in the tank.
- *TSR1*: The Estimator component shall implement a filter with the Kalman algorithm that shall use both input from the fuel tank and the EMS to provide a value of the estimated fuel level that does not deviate 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 since an older value could result in a too big deviation from

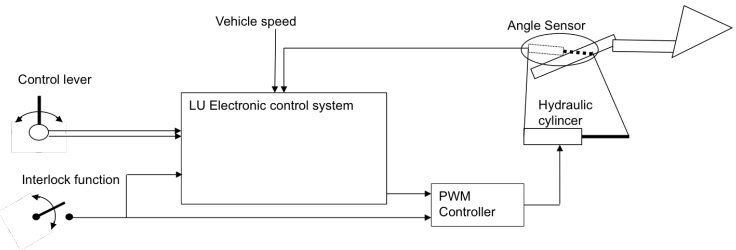

Figure 3.2: Loading Arm Control Unit High-Level Architecture

the actual fuel consumption that may cause deviation in the estimated fuel level value.

3.2 Loading Arm Controller Unit

Wheel loaders are heavy equipment machines used in construction to move material or load material into/onto other types of machinery such as trucks. They are equipped with a lifting arm, which can perform up and down movements that are directly controlled by the operator. Loading Arm Controller Unit (LACU) is a software controller commonly used in wheel loaders for controlling the loading arm of the vehicle. We base LACU on Volvo Construction Equipment demonstrator [67] developed within the SafeCer project[1].

3.2.1 The LACU Architecture

A generic structure of the loading arm item is shown in Fig. 8.1. Based on the inputs from the connected sensors and actuators, LACU issues commands to the hydraulics component to perform the arm movements. LACU is composed of at least three main components, the arm angle sensor monitor, Loading Arm Automatic Positioning (LAAP) and an arm controller that issues the final command to hydraulics.

[1] SafeCer – Safety Certification of Software-Intensive Systems with Reusable Components. ARTEMIS-JU-269265.

3.2.2 The Criticality of the System

Some of the hazards identified for LACU are:

- (H1) unintended movement of the lifting arm, and
- (H2) high-pressure 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 the 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 for bystanders present in the area while high precision movement is required from the machine. On the other hand, the hazard H2 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.

3.3 Summary

In this chapter we have presented two real-world examples that we use in this thesis for performing both exploratory and explanatory case studies. In particular, we mainly use FLES to exemplify the thesis contributions presented in the coming chapters, while we use LACU to evaluate the thesis contributions in two explanatory case studies presented in Chapter 8.

Chapter 4

Contract-driven Assurance

In this chapter, we present the contributions that address the first research goal of the thesis. To achieve the automated generation of assurance arguments from system models, we need to connect the system modelling with the assurance case modelling and evidence management. We first introduce the background information on the connection between assurance case modelling and evidence management introduced by the OMG group in form of a standardised language for modelling. Then, given the background information about safety assurance in Section 2.1 and contract-based design in Section 2.5, we outline the assurance strategy for contract-driven assurance in form of argument patterns that conceptually connect the system with assurance and evidence elements. Then, we present the safety element (component) meta-model that defines the connection between the system, assurance case modelling and evidence management domains. Finally, we present how such a meta-model can be the basis for automated instantiation of the contract-driven assurance argument-patterns.

4.1 Structured Assurance Case Meta-model

Structured Assurance Case Meta-model (SACM) is an Object Management Group (OMG) standard that specifies a meta-model for representing structured assurance cases [24]. The purpose of the standardised meta-model is to provide better portability and exchange of the safety arguments used to represent the assurance cases. In particular, SACM has brought together the assurance case modelling and evidence management domains by connecting them in a single

meta-model. SACM 2.0 consists of argumentation and artifact meta-models. The two meta-models are combined and supported through the common base elements. In the reminder of the section we provide the essential background information about the two meta-models, and clarify their relation to GSN.

4.1.1 SACM Argumentation meta-model

Figure 4.1 shows the SACM argumentation meta-model part (elements represented with solid borders). It defines the basic concepts required to model structured arguments. The argumentation elements in SACM are grouped into single meta-classes where the specific nature of their usage is specified in the attributes of those classes. For example, a generic element *Claim* represents all the different kinds of claims in the argument, and whether the claim is a supporting statement, assumption, justification or context, is specified in the attribute of the element. This portion of the SACM meta-model does not cover artifacts, but only citations to artifacts, which are further detailed in the SACM Artifact meta-model. The artifact citations are captured by the *ArtifactReference* element and cover different kinds of citations such as the ones that can be used as context statements in an argument, citations of away elements or entire modules. The links between the different elements are achieved by the *AssertedRelationship* element, where different asserted relationships are used for different types of connections. Also, the *isCounter* attribute can be used to state that one element is countering another element, which can be used to specify for example counter evidence. Modular argumentation is enabled by capturing arguments in argument packages through the *ArgumentPackage* element. Furthermore, linking different packages is also supported, as well as citation of elements across packages. Since this meta-model captures the basic argumentation elements and their relationships, it can be used to instantiate different compliant meta-models for different argumentation notations such as GSN [27] and Claims-Arguments-Evidence (CAE) [68]. The purpose of such a common meta-model is to facilitate interchange of the structured argumentation documents produced by different tools that use different argumentation notations. Figure 4.1 shows the mapping of the basic subset of GSN elements (represented with dashed borders) to the SACM argumentation meta-model.

4.1.2 SACM Artifact meta-model

The SACM Artifact meta-model specifies artifacts as the main evidentiary of an assurance case. It allows for the assurance case evidence management, which

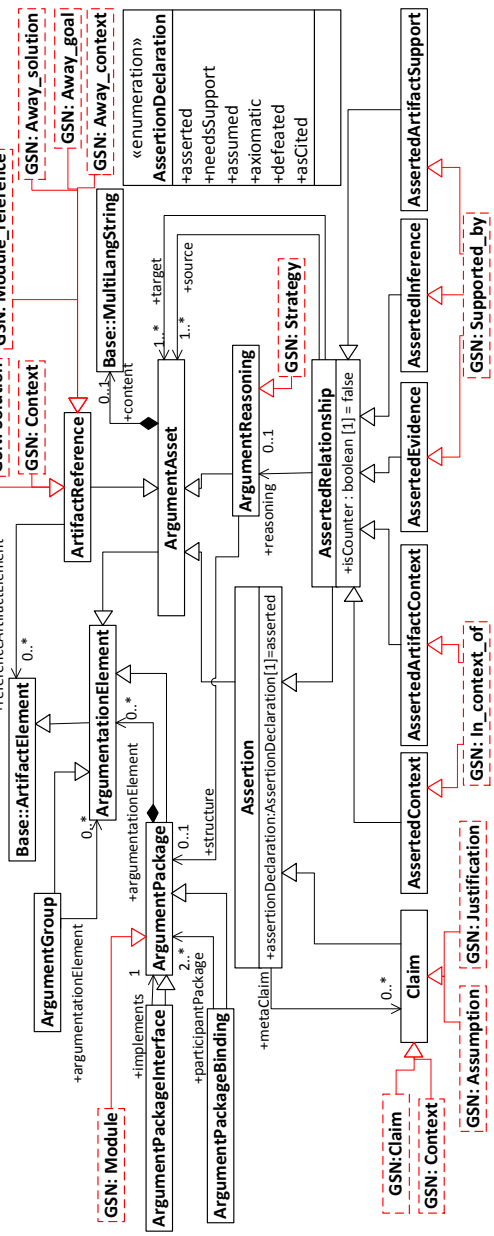

Figure 4.1: A subset of GSN elements (dashed borders) mapped to the SACM argumentation meta-model

includes both evidence description, storage and versioning. As mentioned in Section 2.1, the evidence is one of the main pillars of safety cases alongside the requirements that they support and the argument that connects the two. *Evidence* is information or an objective artefact offered in support of one or more claims [24]. Anything that supports a claim can be referred to as evidence. Evidence is usually based on established facts or expert judgement. Generic examples of evidence in the context of safety cases are test results, system architecture, and tool/personnel competence.

Evidence can be categorised with respect to different characteristics such as nature of support and quality of information it offers [24], or based on the characteristics of the document that is the source of the information [69]. We focus on the categorisation of evidence based on the nature of support it offers [69], i.e., proximity of the evidence to the product it supports, which categorises evidence as immediate, direct and indirect. The *immediate evidence* represents the original artefacts that is being evaluated as evidence such as source code, system architecture, specifications and requirements. The *direct evidence* represents the direct properties of immediate artefacts and is typically sufficient on its own to support a claim, e.g., test results, hazard and failure logic analyses. The *indirect evidence* (also referred to as circumstantial evidence) represents information related to the direct evidence and is typically not sufficient to support a claim on its own, but require introduction of additional evidence. Typical examples of indirect evidence include tool and personnel qualifications as well as development process.

The SACM Artifact meta-model explicitly groups the evidence on artifacts, activities, events, participants, techniques and resources. The modelling of the different kinds of evidence is not only supported when the evidence is available, but SACM also makes it possible to specify the evidence that is not available yet, to indicate that this type of evidence with the particular properties is needed.

4.2 Arguing Contract-driven Assurance

Contracts as pairs of assumption/guarantee assertions about a component are intended to support compositional verification of a composite system [66]. An example of such a system is FLES, introduced in Chapter 3. In the context of FLES, consider the partial requirement decomposition for the hazard H1 (running out of fuel while driving), presented in 3.1.2: the system safety goal SG1 is allocated to FLES, FSR1 as a functional safety requirement addressing SG1

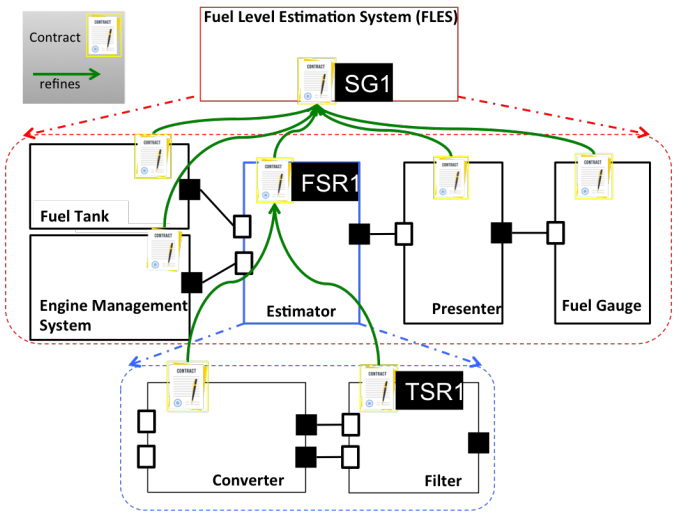

Figure 4.2: An example of a contract specification on different composition levels for Fuel Level Estimation System

is allocated to the Estimator component, and the derived technical safety requirement TSR1 is allocated to the Filter component, as depicted in Figure 4.2.

To support the design of a composite system such as FLES and enable compositional verification, contracts should be defined on each level of composition as shown on the FLES example in Figure 4.2. Such contracts are specified by either formalising the requirement allocated to a component or capturing the component behaviour that implies that the requirement is satisfied. For example, a contract for the Estimator component realising FSR1 can be informally formulated as follows:

A: (fuelLevelSensor within correct range AND fuelLevelSensor does not deviate more than 10% from the actual fuel level value) OR
(fuelRate within correct range AND fuelRate does not deviate more than 1% from the actual engine consumption value AND Tank size within [230-1000])

G: totalFuelLevel does not deviate more than -1% from the actual fuel level value

To guarantee the behaviour that addresses FSR, the contract assumes that either the fuel level sensor value located in the fuel tank of a truck, or the fuel rate value provided by the engine management system uphold certain properties.

While compositional verification of a system using contracts establishes validity of a particular requirement on the system model in terms of contracts, the assurance that the system implementation actually behaves according to the contracts requires some additional reasoning. To capture this reasoning and argue contract-driven assurance, we present two argumentation patterns:

- *"Contract-driven requirement satisfaction assurance"* – this argument pattern deals with the connection of the safety requirements allocated to components and the component contracts, and how such contracts can be used to assure that the requirement is achieved by the component, i.e., that the component successfully implements the requirement with sufficient confidence.

- *"Contract satisfaction assurance"* – this argument pattern deals with the satisfaction of the component contracts in the given context, i.e., whether the contracts are valid with sufficient confidence.

Figure 4.3 presents the contract-driven requirement satisfaction assurance argument pattern. To assure with sufficient confidence that a system satisfies a safety requirement based on the related contracts, we need to provide evidence that the contracts correctly represent the requirement (often said that the contract guarantees formalise the requirement) and evidence that the contracts are valid in the given system context. The validity of a contract greatly depends on its completeness and truthful representation of the actual implementation, e.g., as there may be a missing property in the contract specification that may render the contract invalid in the given context. Hence, besides the contract validity evidence, we also need to assure sufficient confidence in that validity, i.e.,

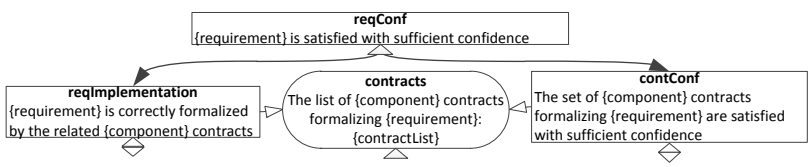

Figure 4.3: Contract-driven requirement satisfaction assurance argument pattern

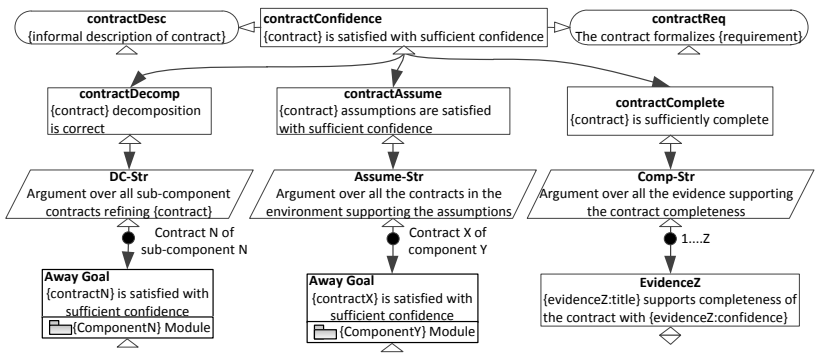

Figure 4.4: Contract satisfaction assurance argument pattern

that the contracts are sufficiently complete. We formulate the sufficient completeness of a contract as an acceptably low likelihood that there are missing properties in the contract specification that may render the contract invalid.

Figure 4.4 presents the contract satisfaction assurance argument pattern. We use satisfaction here to refer to a contract that is validated in a given context, i.e., its consistent with the other contracts and its assumptions are validated in the given context. Furthermore, when we deal with hierarchical systems such as FLES, where contracts are defined on each hierarchical level with well defined decomposition conditions, then to argue that the composite component behaves according to the contract, we should explicitly argue over the component decomposition. To assure satisfaction of a component contract with sufficient confidence we devise the supporting argument with three goals:

- the contract specification is sufficiently complete (the goal **contractComplete**);
- the environment (made of the other components and the system context) meets the needed assumptions of the component (the goal **contractAssume**);
- the internal subcomponents of the component correctly refine the component contract, i.e., the composite component decomposition is correct (the goal **contractDecomp**);

The confidence in satisfaction of a contract depends on both the internal components and their contracts, as well as the neighbouring components and

their contracts. Hence, both the assurance of the component decomposition and its relation to the environment in which it is deployed are addressed in a similar way in the *contractDecomp* and *contractAssume* argument branches. In both cases we refer to the argument modules that address the confidence in those supporting contracts, be it sub-component or external contracts. The evidence for those two branches is supported by the various contract checks that can be performed during formal analysis, e.g., contract consistency, validity and refinement checking as done in the OCRA tool introduced in Section 2.5.1. Conversely, the evidence for the completeness branch is generally external to those formal analysis results as that evidence aims at increasing confidence that the specification itself on which the formal analysis was done is sufficiently complete.

4.2.1 Contract-driven assurance case structure

As mentioned in Section 2.1, an assurance case can be structured into modules that encapsulate argument structures focusing on a particular goal. The contract-driven assurance case structure assumes that an assurance module is created for each system component where the contract-satisfaction patterns are instantiated to argue over each contract of the component. A separate module that depends on those component modules is created for arguing over satisfaction of requirements. The contract-based requirements satisfaction pattern is used to build that argument module and connect that module to the corresponding component modules. The basic structure of the contract-driven assurance case is shown in Figure 4.5.

4.2.2 Contract-driven assurance supporting evidence

To drive the system assurance using contracts we associate with each contract a set of assurance assets. Those assets can be the above mentioned evidence that increase confidence that the component (i.e., the implementation of the contracts) behaves according to the contract, i.e., that the component deployed in any environment that satisfies the contract assumptions exhibits the behaviours specified in the corresponding contract guarantees. 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 capture all the needed properties of the component and the environment,

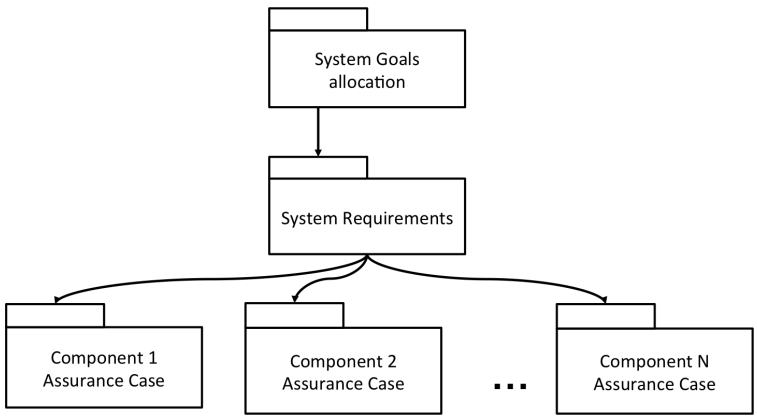

Figure 4.5: Contract-driven assurance case structure

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.

These three evidence categories represent the three pillars for assuring confidence in contract-driven assurance. The correctness evidence is used in the contract-driven requirement satisfaction argument, while the completeness and consistency are used for the contract satisfaction arguments. The consistency evidence represents any formal analysis results that establish the validity of a particular property of the contract specifications, such as for example refinement analysis results. While consistency evidence is used to support the contract decomposition and assumption satisfaction goals, the completeness evidence is presented separately.

4.3 Safety Element Meta-Model

To connect the assurance and system modelling domains, we present the Safety Element Meta-Model (SEMM) in Figure 4.6. A *safety element* is a component that besides its implementations, is composed of the safety requirements that specify the system safety-relevant demands on the component, contracts that capture those demands and describe their implementation, and evidence that

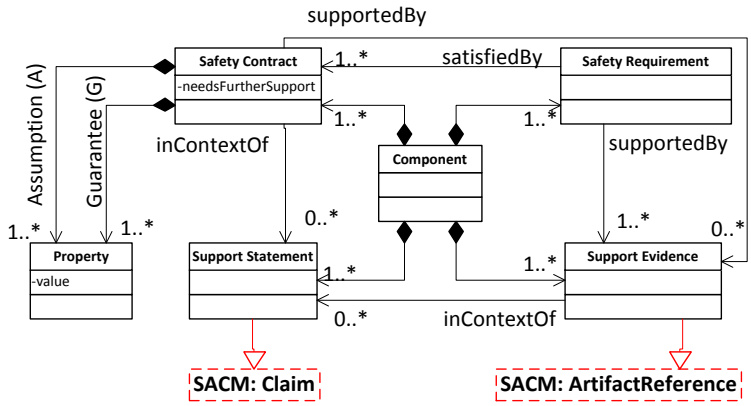

Figure 4.6: Safety Element Meta-model and its relation to SACM

supports confidence in the behaviour of the component. Each safety requirement is satisfied by at least one contract, and each requirement and a contract can be supported by evidence. Since the aim of safety assurance is to build structured arguments that clearly communicate confidence, we include as a part of such a safety element also the supporting statements that aim to increase the clarity of the different contracts, requirements and evidence, as well as their interconnections. The *Safety Contract* metaclass is furthermore enriched to include explicit support for specifying that the contract is not fully validated, i.e., only partial evidence is provided with the contract and additional evidence should be provided. This is achieved with the *needsFurtherSupport* attribute. A contract is composed of two properties, namely assumptions denoted with A and guarantees denoted with G.

The aim of SEMM is to bridge the system modelling with the assurance case modelling. In SEMM we do not detail the compositional nature of systems and composite components or fine-grained assurance support, but focus on a component as a safety element, and its relations to the essential assurance assets. We strengthen the connection between the assurance case about a system and its SEMM compliant system model by relating both to an Evidence Container compliant to the SACM Artifact meta-model.

As mentioned in Section 4.1.2, based on the nature of the evidentiary support we distinguish between immediate, direct and indirect evidence. The evidence that is used to support the safety contracts qualifies as the direct evi-

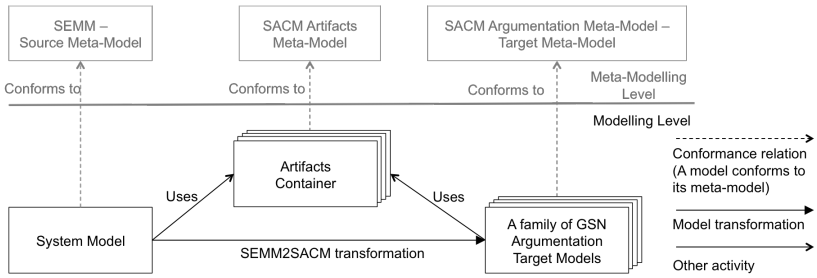

Figure 4.7: Connection between the SEMM and SACM meta-models

dence. Regardless of the type of the evidentiary support, all evidence is managed by the SACM-compliant Evidence Container. SACM allows for detailing the nature of the support of evidence related to a contract through the artifact properties, so that we can distinguish between evidence that supports contract consistency, completeness and correctness, as discussed in Section 4.2. Depending on the indicated type of the supporting evidence, the evidence is used accordingly in the corresponding assurance. For example, the correctness evidence related to a contract with respect to a corresponding requirement is used to assure that the requirement is correctly formalized/realised by the contract.

The connection to the SACM Argumentation and Artifact meta-models introduced in Section 4.1, is achieved through the supporting statement and evidence elements, which correspond to the SACM Claim and Artifact Reference elements, respectively. The connection between SEMM and the SACM Argumentation and Artifact meta-models is shown in Figure 4.7. Since the same artifacts sources are referred within both the assurance case and the system model, automated transition between the assurance and system domains is possible. We use automated model to model transformations to generate GSN argumentation from a SEMM compliant specification. In the reminder of the section we detail the model to model transformation.

4.3.1 SEMM to SACM transformation

Transforming the information captured within a SEMM compliant model to a GSN argumentation model compliant with the SACM argumentation meta-model results in a set of contract satisfaction argumentation-fragments. Those fragments represent particular instantiations of the argumentation pattern presented in Figure 4.4. These fragments can be organised in different ways,

Algorithm 1 M2M Transformation from SEMM to SACM

```
SEMM2SACM(in SEMM, out SACM){
  for each SafetyContract sc in SEMM do
    sc2claim(in SEMM::SafetyContract, out SACM::Claim);
    for each SupportStatement s in context of sc do
      s2claim(in SEMM::SupportStatement, out SACM::Claim);
    end for
    addAssumeGoal(in SEMM::SafetyContract, out SACM::Claim);
    for each Assumption Property a in sc do
      a2claim(in SEMM::SafetyContract, out SACM::Claim);
      a2away (in SEMM::SafetyContract, out SACM::ArtifactReference);
    end for
    addCompleteGoal(in SEMM::SafetyContract, out SACM::Claim);
    for each SupportEvidence se supporting sc do
      se2claim(in SEMM::SupportEvidence, out SACM::Claim);
      se2Sol(in SEMM::SupportEvidence, out SACM::ArtifactReference);
      for each SupportStatement ss in context of se do
        ss2claim(in SEMM::SupportStatement, out SACM::Claim);
      end for
    end for
    if decomposition via refinement defined then
      addDecompGoal(in SEMM::SafetyContract, out SACM::Claim);
      for each refining contract rc of sc do
        rc2away(in SEMM::SafetyContract, out SACM::ArtifactReference);
      end for
    end if
  end for
}
```

for instance to populate the contract-driven requirement satisfaction assurance pattern, described in Figure 4.3. Regardless of the top-level organisation of the generated argument-fragments, their population follows a set of common transformation rules.

The common transformation rules are summarised by means of pseudo-code in Algorithm 1. The *SEMM2SACM* model-to-model (M2M) transformation generates a set of argument-fragments, one for each safety contract. The algorithm is composed of three main steps and one optional step, which are

performed for each of the contracts of a component:

- **Step 1 –** The contract is transformed to the top claim in *sc2claim*, which is clarified via the related support statements in *s2claim*. The top claim is then decomposed via *addAssumeGoal*, *addCompleteGoal*, and optionally if the component decomposition via contract refinement is defined in the particular system, then *addDecompGoal* is also performed.
- **Step 2 –** Populate the subgoal created with *addAssumeGoal*. For each of the contract's satisfied assumptions a goal is created in *a2claim* to argue over satisfaction of the assumption. The goal that presents the satisfaction of the contract satisfying the assumption is associated to the assumption goal via an away goal in *away2a*.
- **Step 3 –** Populate the subgoal created with *addCompleteGoal*. For each support evidence associated with a contract create a goal in *se2claim*, and then support it with the artifact reference in *se2Sol*. Finally, attach the related support statements in *ss2claim*.
- **Step 4–** Given that the component decomposition is defined via contract refinements, populate the subgoal created with *addDecompGoal*. For each refining contract create an away goal in *rc2away* to the claim presenting the confidence in the refining contract.

We summarise the basic transformation rules in more detail:

- *sc2claim*: creates the top-level goal of the contract satisfaction argument pattern that the contract is satisfied with sufficient confidence;
- *s2claim*: creates and associates context statements with the top-level goal regarding the informal description of the contract and its relation to requirements;
- *addAssumeGoal*: creates the generic *contractAssume* goal from the contract satisfaction argument pattern that the contract assumptions are satisfied with sufficient confidence and associates it with the top-level goal;
- *a2claim*: creates a subgoal of the *contractAssume* goal claiming that the particular assumption is satisfied with sufficient confidence;
- *away2a*: creates an away goal pointing to the contract that guarantees satisfaction of the above argued assumption and associates it to the corresponding assumption goal;

- *addCompleteGoal*: creates the generic *contractComplete* goal from the contract satisfaction argument pattern that the contract is sufficiently complete and associates it with the top-level goal;
- *se2claim*: creates a subgoal of the *contractComplete* goal to present how particular evidence artifact supports the contract completeness;
- *se2Sol*: creates and associates the solution element with the corresponding artifact specific goal;
- *ss2claim*: creates and associates the supporting context statements with the artifact specific goal clarifying how the artifact supports the completeness of the contract;
- *addDecompGoal*: creates the generic *contractDecomp* goal from the contract satisfaction argument pattern that the contract decomposition is correc and associates it with the top-level goalt;
- *rc2away*: creates an away goal pointing to the satisfaction argument of the contract specified in the decomposition and associates it with the *contractDecomp* goal;

4.4 Summary

In this chapter we have presented how contract-based design and development can be the basis for building an assurance case. In particular, we have presented a couple of argumentation patterns describing how to build confidence in the system developed and design using contracts. Then, we have presented a meta-model that allows for capturing the needed assurance information during system modelling. Finally, we have presented how the safety-assurance aware system models can be used to automatically instantiate the proposed contract-driven assurance argumentation patterns.

Assuring that a system is sufficiently safe involves many stakeholders in the system development. For example, when developing a safety-critical system using the contract-based design paradigm then the activities such as contract specification, formal analysis and safety assurance using the results of such analysis are not necessarily done by the same person. To build a safety argument and connect the evidence with the safety goals, the safety assurance engineer relies on understanding the evidence. The idea with connecting contracts with assurance elements such as claims, evidence and context statements

is that the system engineer specifying the contracts can at the same time clarify and support the contracts with some additional statements as well as the artefacts used to obtain them or verify them. Capturing the additional information during the system modelling allows the safety assurance engineer to instantiate argumentation patterns with pre-prepared information. The generated argument-fragments represent only a portion of the overall argument and can be seen as the skeleton that the overall argument can be built upon. Even after the automated argument-fragment generation, the need for further manual tailoring of the argument remains. The semi-automated nature of such generation of an argument preserves the possibility for customised tailoring of the argument, while enabling benefits of getting a head start by automated generation of parts of the argument.

Automation in safety argumentation is often said to be dangerous as it reduces the critical thinking involved in the process, which in turn may negatively impact the quality of the resulting argument. In this thesis, we argue that automated instantiation does not mean that the critical reasoning by the engineer when building the argument has been lost, but just that part of the effort for building the confidence in the specification was moved to the system engineer. What remains for the assurance engineer is to review whether the specification fits the requirements and whether the system engineer has managed to assure the confidence in the specification.

Chapter 5

Contract-Driven Reuse for Safety-Critical Systems

In this chapter, we first look at some examples of reusable safety-relevant components from our real-world examples in Chapter 3. Then, to provide greater support for reuse of safety-relevant components, we present an extension of the assurance-aware contract based design in form of strong and weak contracts. Finally, we align the assurance and reuse aware contract-based design with the safety process mandated by ISO 26262 and its Safety Element out of Context concept for reuse.

5.1 Examples of reusable safety-relevant components according to ISO 26262

As mentioned earlier, reuse of components in safety-critical systems is not sufficient without reuse of the accompanying assurance assets, i.e., information needed for assuring adequate levels of safety. Since what is needed for assurance is influenced by the domain specific safety standards, we turn to the standards for initial guidance on reuse. In particular, we turn to the automotive ISO 26262 standard introduced in Section 2.3 and its SEooC notion for development and reuse of safety-relevant components promoted, described in Section 2.3.1. Regardless of our focus on the automotive domain, the very concepts for reuse introduced through the SEooC notion as well as the contract-

based design are generic enough to be applicable in other domains as well. Given the system descriptions in Chapter 3, we highlight two examples of SEooC candidates in each system:

- **Fuel Level Estimation System (FLES):** FLES estimates fuel in the tank of a truck and displays the estimate to the driver. Since FLES represents an item, it cannot be a SEooC by itself, but the *Estimator* component makes a good SEooC example since the estimation process needs to be tailored for the specific environment. For example, the size and shape of the tank in a particular truck model influences the choice of the filtering algorithm and its settings. Hence, Estimator should be developed to work in these different environments. Furthermore, any differences in its behaviour due to its different execution in different environments should also be known.

- **Loading Arm Controller Unit (LACU):** LACU facilitates the driver of wheel-loader to perform movements of the loader arm, either directly via control lever or indirectly, via automatic positioning button. Since the entire LACU cannot be a SEooC, the Loading Arm Automatic Positioning (LAAP) component is a good example of a SEooC. LAAP implements the functionality to automatically move the arm to a particular position. In different wheel-loaders it behaves differently, hence the component is developed with some configurable parameters such as the ground speed limit after which the automatic positioning cannot be used, or whether the position to which the arm is moved automatically is pre-defined (a default position), or if it can be dynamically set by the operator.

5.2 Strong and weak contracts

Reuse is intrinsic to contract-based design. It enables checking if a component can be reused in a particular system, i.e., whether the system meets its demands and whether the component meets the demands of the system. The support for reuse in contract-based design has been mainly focused on components (i.e., implementations of contracts) and not reusable components as implementations of a set of contracts for different environments that may or may not be satisfiable together. Reusable components such as SEooC are often characterised with parameters that are used to tailor the behaviour of the component in the different settings in which the component is reused. For example,

if we consider the LAAP component for automatic positioning of the loading arm as a reusable component. It can be deployed in vehicles that have a fixed position for automatic positioning, or it can be deployed in vehicles that have a dynamic positions for automatic positioning that user can set. Since unintended arm movement is one of the hazards for such vehicles, in the first environment with fixed positioning no special demands need to be made on the accuracy of the fixed position, as it never changes, which can be checked internally within the LAAP component. But if the position is dynamic, then LAAP needs to rely on its environment to provide a position value with sufficient accuracy. This leads to having two different contracts about the same behaviour of the LAAP component, but with different assumptions, which means that we can have one set of correct environments for all the component contracts, but there are environments that are correct for some, and not correct for other contracts. Since contracts capture behaviour of the component, they should also support some variability in capturing such behaviours. To address such need for variability at the contract level, we make a distinction between strong and weak contracts. On the one hand, the strong contracts are those whose assumptions should be met by every context in which the component is reused, hence its guarantees are always offered by the component. On the other hand, the weak contract assumptions do not need to be satisfied by every context in which the component is reused, but when they are met, only then the component offers the corresponding weak guarantees.

We formally describe the strong and week contracts in terms of environments in the context of the trace-based contract framework [66]: for a component S described with a set of strong contracts $\xi_S(S)$ and a set of weak contracts $\xi_W(S)$, we say that an environment E is a *correct environment* of S if: $\forall(A,G) \in \xi_S(S), E \subseteq [\![A]\!]$, i.e., for an environment of S to be correct, it must satisfy the assumptions of all the strong contract of S. We denote with $\mathcal{E}(S)$ all the correct environments of S. Such correct environments may or may not satisfy the assumptions of the weak contracts of S. While this provides some flexibility in specification of contracts, it may also mean that some weak contracts may never be validated in any of the correct environments e.g., if a weak contract is contradicting a strong contract. For S not to contain such unnecessary weak contracts we require that each weak contract of S has at least one correct environment that satisfies its assumptions, i.e.,: $\forall(A,G) \in \xi_W(S), \exists E \in \mathcal{E}(S), E \subseteq [\![A]\!]$.

The problem with specifying such contracts is that if we try to check refinement by considering all the specified weak contracts, the check will fail since a single environment might not be able to meet the assumptions of all the weak

contracts. To overcome this problem without redefining the notion of contract refinement, we can either

- filter the weak contracts before checking the refinement, such that only those weak contracts whose assumptions are met by the current environment are included in the refinement check; or
- transform the weak contracts in a different format such that refinement can be performed.

Weak Contract Filtering

While a SEooC is described with sets of both strong and weak contracts, when instantiated to a particular correct environment E then, for the purpose of refinement check, it is enough to describe the SEooC instantiation with a subset of contracts that are applicable in the environment E. Given a SEooC component S and its instantiation S' in a correct environment $E \in \mathcal{E}(S)$, the set of contracts of S' denoted with $\xi(S')$, which contains the contracts considered during refinement check, is a union of all the strong contracts from $\xi_S(S)$ and only those weak contracts from $\xi_W(S)$ whose assumptions are satisfied by the environment E.

Weak Contract Transformation

Instead of filtering only some weak contract to perform the refinement check, the refinement check could be performed if the weak contracts are transformed such that they do not impose restrictions on the environment. This can be done if the weak contract assumptions are relaxed. For a weak contract $C = (A, G)$ of a component S, a relaxed counterpart of this weak contract would be $C' = (true; A \implies G)$, where $true$ represents an assertion satisfied by all environments. The relaxed counterpart has relaxed assumptions, hence it differs from the corresponding weak contract in terms of environments, but they are the same from the perspective of implementations. Since the assumption of C' is satisfied by every correct environment of S, it can be regarded as a strong contract. Since any contract that is refined by C is also refined by C', either form can be used for the sake of checking refinement of a weak contract. If we have a set of weak contracts and we transform them to their relaxed form and conjunct them to a single contract by conjunction of their guarantees, then any contract that is refined by at least one of those weak contracts is also refined by the conjuncted contract. The SEooC instantiation in a particular context

does not require contract filtering in this case, but the in-context component can inherit both strong and weak contracts. Since the refinement check by considering all the strong and weak contracts would fail in case of two weak contracts that do not share the same correct environments, we transform the weak contracts to the appropriate format described as follows: given a SEooC component S and its instantiation S' in a correct environment $E \in \mathcal{E}(S)$, the set of contracts of S' denoted with $\xi(S')$, which contains the contracts considered during refinement check, is a union of all the strong contracts from $\xi_S(S)$ and the conjuncted contract of all the weak contracts in their relaxed form from $\xi_W(S)$.

Although this approach allows all the contract specifications to be used for checking the refinement, it does not reveal which weak contracts are relevant in the environment E, i.e., assumptions of which weak contract are satisfied by E. Not knowing which weak contract is relevant in the current environment means that we do not know which weak contract and its assurance assets we should use in the assurance case. For the sake of reuse we still need to check which weak contracts are relevant in the environment E.

5.2.1 Contract variability in SEMM

In Section 4.3 we introduced the SEMM metamodel to connect the system with assurance case modelling. SEMM takes contract-based design one step ahead in providing greater support for reuse of safety elements, where the reuse includes not only the components and their contracts, but also the accompanying assurance information. As discussed earlier in this chapter, while this is sufficient for supporting reuse of components tailored for particular use, it is not enough to deal with reuse of configurable components that have different behaviours in different environments, and consequently, different assurance information relevant in those environments. For such components, SEMM needs to be extended with the strong and weak contracts such that it is possible to model the configurable components made for reuse. Furthermore, it should be possible to capture the relevant contract selection in a given environment.

Figure 5.1 represents the SEMM extension with the strong and weak contracts that allows for modelling of out-of-context configurable components such as the SEooC examples in Section 5.1. We refer to this extended SEMM version also as SEooCMM, explicitly expressing its support for SEooC modelling. Since the supporting assurance information, including the different statements, requirements and evidence, are connected to safety contracts, by identifying the relevant contracts, we also identify the relevant assurance in-

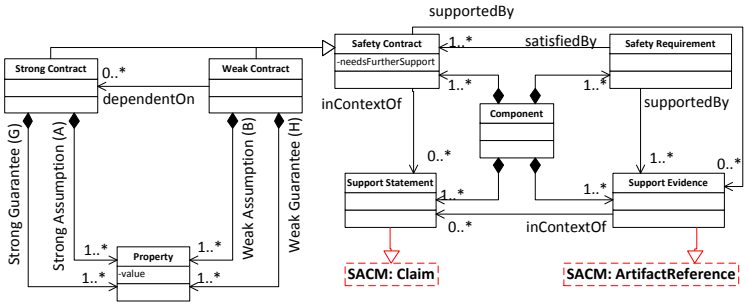

Figure 5.1: Safety Element (out-of-Context) Meta-model with strong and weak contract extension

formation. Knowing which information that we have reused is relevant in the given environment allows us to automate generation of context-specific argument-fragments by instantiating the requirements and contract satisfaction patterns using only the information identified as relevant.

5.3 Contract-aware SEooC development and reuse

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

5.3.1 Safety Contracts Development Process

As mentioned in Chapter 1, the nature of safety being a system property and the dangers of non-systematic reuse hinder reuse of safety components 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 components. 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

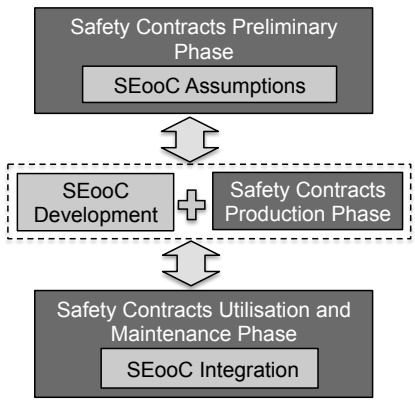

Figure 5.2: *Safety contracts development* and *SEooC development and integration* processes combined

phases: (1) *Preliminary* safety contracts, (2) Safety contracts *production*, and (3) Safety Contract *utilisation and maintenance*. The alignment of the safety contract and the SEooC development phases is shown in Fig. 5.2. While the first two safety contract phases support the two out-of-context phases of the SEooC development (Fig. 2.4), the third safety contract phase supports SEooC integration in context of a particular system. The first safety contract phase includes the capturing of SEooC assumptions. The second contract phase is performed together with the corresponding SEooC development phase, while the third contract phase includes support for integration of SEooC in a particular system. 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

This phase should be performed before the development of the item/component for which the contracts are being established. The phase constitutes of the following contract-specific activities:

- *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., accu-

racy 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 [70]. 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 (assumed) HW/SW safety requirements*: As mentioned in Section 8.3.4, the safety contracts should capture information needed to satisfy the safety requirements allocated to the corresponding safety component. For example, supporting each derived SW safety requirement allocated to a software component 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

This phase should be performed during the component development stage and the following verification and validation activities on the component level. The phase constitutes of the following activities:

- *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 component development stage, such contracts need to be finalised once the actual behaviour of the component (or a more accurate approximation) can be established. For example, when more accurate information about the actual accuracy of an algorithm, timing behaviour, or memory footprint of the component 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. The evidence related to the contracts consistency is generated by checking whether the contracts are free of contradictions (e.g., strong and weak contracts of the component do not make any contradicting assumptions on the same property). The confidence in completeness of the information captured within the contracts can be for instance increased through the evidence from which the contract is derived. 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. 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

This phase is performed in context of a particular system and the activities related to contract utilisation and maintenance can be performed at different stages of the system lifecycle. For instance, the safety contracts can be utilised for:

- Supporting architectural design of safety-critical systems.
- Selection of components based on the safety-relevant behaviours captured in the safety contracts.
- Integration of a component in an existing system, as one of the main roles of contracts is to promote independent development of components and their easier integration via contract-based verification.
- Contract-based safety assessment activities in form of *contract-based artefacts generation*. Contracts can be utilised for generation of different

safety case artefacts such as safety case argument-fragments.

The supporting activity for the contract development and utilisation is *contract maintenance*. For example, 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.

5.3.2 SEooC Development with Safety Contracts

SEooC development starts by capturing the system-level assumptions (Fig. 2.4). Simultaneously, the preliminary safety contracts phase is initiated, as described in the Section 2.3. 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 (e.g., manually) to the actual safety requirements allocated to the component, and contracts are used to verify that the assumptions captured during the SEooC development are satisfied (provided that the contracts are established for the rest of the system). The contract production phase continues in-context to capture the behaviours of the SEooC that could not be established out-of-context. In case of assumptions 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 component is generated to show the satisfaction of the safety requirements through the satisfaction of the associated safety contracts.

5.4 Summary

Contract-based design inherently supports reuse of components in form of contract implementations. But to fully understand the behaviour of a component

and its safety implications, the context in which that behaviour is exhibited needs to be known. While component contracts represent a way of capturing a part of that context, additional context information is typically needed when dealing with safety-relevant components. In this chapter we have presented contract-driven support for fine-grained reuse of safety-relevant components developed specifically for reuse in different systems and independently of a specific system. In particular, we have used the notion of strong and weak contracts to manage variability on the specification level. We have included that contract variability in the Safety Element Meta-Model by interconnecting strong and weak contracts with assurance information. Through this connection of system and assurance case modelling information, the variability on the contract level allowed for variability on the assurance level as well.

Just as any safety-critical system, such components should also follow specific safety process and fulfil particular requirements mandated by the relevant safety standards. We worked in the context of the automotive domain that described high-level principles for development, reuse and integration of such components in the form of the Safety Element out-of-Context notion. The idea with such components is that the implementation comes together with partial safety assurance information, which the integrator can utilise to speed up the integration and reduce the cost of its safety assurance. In this chapter we have proposed an extended assurance and reuse -aware contract-based design as a way to achieve reuse, integration, and assurance of Safety Element out-of-Context components. In particular, we have proposed the process of the assurance and reuse aware contract-based design and development and aligned it with the mandated Safety Element out-of-Context principles from ISO 26262.

Chapter 6

Contract-driven assurance and reuse based on Compositional FLA Results

In this chapter we present a method called FLAR2SAF that uses Failure Logic Analysis Results (FLAR) to generate Safety case Argument-Fragments (SAF). We first present the essential background information FLAR2SAF is based on, and then we present FLAR2SAF. More specifically, in FLAR2SAF we combine the contract-driven assurance and reuse with an existing failure logic analysis. Then, we present how such a combination of assurance aware contract-based design and failure logic analysis can be used to instantiate an existing argument pattern for addressing failure behaviour of components [71].

6.1 COTS Aware Fault Propagation Analysis and Argumentation

In this section we present a brief background of the Commercial Off The Shelf (COTS) driven development as well as CHESS-FLA that supports failure logic analysis of independently developed components. Then, we recall the argumentation pattern for assuring the *Absence of Hazardous Software Failure Mode* of "type" value, which can be analysed using CHESS-FLA.

6.1.1 COTS-based Safety-Critical Development

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 process. As Figure 6.1 depicts, the top-down and bottom-up approach meet in the grey zone. Initially a top-down approach is carried out. The typical safety process starts with hazards identification which is conducted by analysing 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 and decomposed onto the architectural components, as a basis for designing mitigation means. Safety requirements are assigned with Safety Integrity Levels (SILs) as a measure of quantified risk reduction. Iteratively and incrementally the system architecture is changed until a satisfying result is achieved (i.e. no intolerable behaviour at system level). While the system safety process derives the required safety requirements, COTS are developed to meet certain safety requirements deemed relevant for the systems in which that component may be used. The grey zone in Figure 6.1 is the meeting point of the two processes, one carried out for the system and the other for the component development. More specifically, once the system 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. Contract-based design inherently provides support for such COTS selection process. With its support for independent development of components, contract-based design enables checking whether the independently developed component, such as COTS, fulfils the safety requirements imposed by the particular system in which the component is reused. By checking a requirement against a set of COTS components enriched with contracts, it is possible to identify which of the components, if any, meet the particular requirement.

6.1.2 CHESS-FLA within the CHESS toolset

CHESS-FLA [72] is a plugin within the CHESS toolset that includes two Failure Logic Analysis (FLA) techniques:

- Fault Propagation and Transformation Calculus (FPTC) [13] — a compositional technique to qualitatively assess the dependability of component-based systems, and

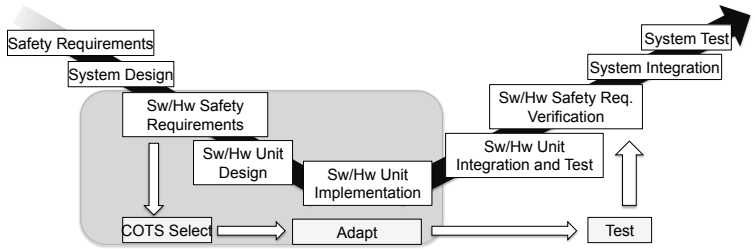

Figure 6.1: Safety-critical system development/COTS-driven development

- A Formalism for Incompletion, Inconsistency, Interference and Impermanence Failures' Analysis (FI^{4}FA) [73] — an FPTC extension that allows for analysis of mitigation behaviour in the specific context of transaction-based computations.

In this thesis we limit our attention to the FPTC technique, which allows users to calculate the behaviour at system-level, based on the specification of the behaviour of individual components.

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 failure modes (occurring on output ports) to combinations of input failure modes (occurring on input ports). These behaviours can be classified as:

- a source failure (i.e., a component generates a failure due to internal faults),
- a sink failure (i.e., a component is capable to detect and correct a failure received on the input),
- propagational failure (i.e., a component propagates a failure it received on the input to its output), and
- transformational failure (i.e., a component generates a different type of failure from the input failure).

In FPTC analysis, 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.

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

Figure 6.2: FPTC syntax supported in CHESS-FLA

The syntax supported in CHESS-FLA to specify the FPTC rules is shown in Figure 6.2.

The input/output behaviour of a component developed out-of-context can be captured by FPTC rules. Figure 6.3 shows a running example that we will use throughout the chapter. The example consists of the Estimator component that takes a sensor value and a single parameter as inputs, and provides the estimated sensor value as output. Such a component can be used when the sensor values are expected to fluctuate frequently, e.g., a sensor for estimating liquid fuel level in the tank of a vehicle. The Converter component converts the sensor value based on the input parameter, while the Filter component normalises the sensor value and mitigates coarse/great value failures. The FPTC rule describing the specific Filter failure behaviour can be specified as: $I1.valueCoarse \rightarrow O1.noFailure$. This example demonstrates the sink behaviour of the Filter component (Figure 6.3) and should be read as follows: if the component receives on its input port I1 a coarse (i.e. clearly detectable) value failure (a failure that manifests itself as a failure mode by exceeding the allowed range), it generates no failure on its output port O1.

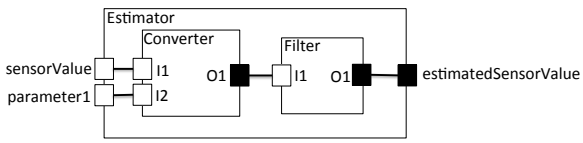

Figure 6.3: Running example

To use the FPTC rules of an individual component for FPTC analysis in a specific system, all possible failure modes that have been considered in the particular system must be considered by the FPTC rules of the component. Since the list of failure modes is not fixed, it can be customised for different systems. Moreover, since specifying all the failure combinations for a component with

a greater number of input ports is tedious and error-prone, it is not necessary to specify rules for all the combinations if there is a default interpretation of such missing rules. One such interpretation is that all missing combinations will simply behave as propagators, considering that this is the worst-case scenario [13]. For example, if the set of FPTC rules for the Filter component does not consider late failure mode on the input port I1, according to this interpretation the late failure on I1 would result in late failure on the output port O1.

Another way to reduce the amount of explicitly specified rules for all the failure mode combinations is through the *wildcard* keyword on an input port, which is used to indicate that the output behaviour is the same regardless of the failure mode on the corresponding input port. For example, the omission failure mode on the output of the Converter component (Figure 6.3) occurs if the I1 input port exhibits omission, regardless of the state of the I2 input port. Instead of writing a set of FPTC rules combining the omission on I1 with all the different considered failure modes on the I2 input, a single rule with a wildcard keyword can be used to cover all the different failure modes on the I2 port, e.g.: "$I1.omission, I2.wildcard \rightarrow O1.omission$".

6.1.3 Absence of Hazardous Software Failure Mode Argumentation Pattern

As mentioned in Section 2.1.2, GSN can be used to represent the individual elements of any safety argument and the relationships between these elements. Figure 6.4 presents the argument pattern for Absence of Hazardous Software Failure Mode (HSFM) [71] of the value type. The argument starts with the goal *AbsHSFMValue* claiming the absence of the HSFM of type value in the software at hand. This goal is further clarified with several context statements defining the software itself, its functionality that contributes to this HSFM, the definition of the HSFM, causes that lead to it and the safety requirements that address them. The top goal is further decomposed using the strategy *ArgFailureMech* that divides the argument based on the failure mechanisms, given that different failure mechanisms are identified for addressing all the known causes of HSFM. The failure mechanisms are classified into three categories:

- Primary failures within a Contributory Software Functionality (CSF) that can cause the failure (the goal *ABValPrimary*). The goal argues that the result produced by the CSF is within the safe bounds;
- Secondary failures relating to other components within the system on

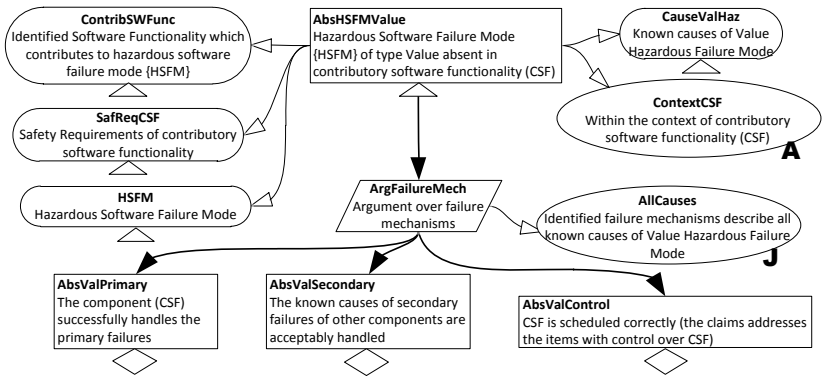

Figure 6.4: Absence of Hazardous Software Failure Mode argumentation pattern for type value failure [71]

which the CSF is dependent (the goal *ABValSecondary*). The goal argues that those failures in the environment of CSF that could lead to CSF producing a failure mode are acceptably handled; and

- Failures caused by items controlling the CSF e.g., the scheduler (the goal *ABValControl*).

6.2 FLAR2SAF

In the heart of FLAR2SAF stands the idea that we can use 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 coarse value failures on the output of Estimator are considered hazardous, then the corresponding argument-fragment should argue that the *valueCoarse* failure mode is sufficiently handled in the context of the particular system and attach supporting evidence for that claim.

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

1. Model the component architecture in CHESS-FLA;

2. Specify failure behaviour of a component in isolation using FPTC rules;

3. Translate the FPTC rules into corresponding safety contracts and attach FPTC analysis results as initial evidence (model compliant with SEooCMM);
4. Support the contracts with additional V&V evidence and strengthen the contract assumptions accordingly;
5. Upon component selection and satisfaction of the strong and the relevant weak safety contracts, depicted in Figure 6.1 in Section 6.1.1:
   (a) Perform FPTC analysis on the system level;
   (b) Translate the results of FPTC analysis to system-level safety contracts;
   (c) Support and enrich the contracts with additional V&V evidence;
6. Use the contract-driven assurance methodology to semi-automatically instantiate the HSFM argument pattern presented in Section 6.1.3 (SACM compliant).

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.

6.2.1 Contractual interpretation of the FPTC rules

In this section we focus on the step of translating the FPTC rules to safety contracts. We use the running example (Figure 6.3) to explain the translation process and provide a set of steps that can be used to perform the translation. In Table 6.1 we have FPTC rules specified for the subcomponents of the Estimator component, and the calculated Estimator FPTC rules. When either of the inputs sensorValue (sV) or parameter1 (p1) exhibit omission failure, the Converter propagates the failure further to the Filter component, which propagates further omission failure to the estimatedSensorValue (eSV) output of the Estimator component. While Converter propagates valueCoarse failures as well, the Filter component mitigates these failures and acts as a sink by transforming them to noFailure. The FPTC analysis of the Estimator component indicates that if omission occurs on any of its input ports, the component propagates the omission failure to the output, while it mitigates any valueCoarse failures that may occur on the input ports.

Table 6.1: FPTC rules of the Estimator, Converter, and Filter components

Component	FPTC rules
Converter:	$I1.omission, I2.wildcard \rightarrow O1.omission$; $I1.wildcard, I2.omission \rightarrow O1.omission$; $I1.valueCoarse, I2.noFailure \rightarrow O1.valueCoarse$; $I1.noFailure, I2.valueCoarse \rightarrow O1.valueCoarse$; $I1.valueCoarse, I2.valueCoarse \rightarrow O1.valueCoarse$;
Filter:	$I1.valueCoarse \rightarrow O1.noFailure$; $I1.omission \rightarrow O1.omission$;
Estimator:	$sV.omission, p1.wildcard \rightarrow eSV.omission$; $sV.wildcard, p1.omission \rightarrow eSV.omission$; $sV.valueCoarse, p1.noFailure \rightarrow eSV.noFailure$; $sV.noFailure, p1.valueCoarse \rightarrow eSV.noFailure$; $sV.valueCoarse, p1.valueCoarse \rightarrow eSV.noFailure$;

Three different types of 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., the second FPTC rule of the Filter component) and rules that describe behaviours that mitigate a failure (e.g., the first FPTC rule of the Filter component). 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 $\langle B, H \rangle_{Estimator-3}$ shown in Table 6.2, guarantees that eSV will not exhibit omission if both inputs sV and p1 do not exhibit omission failures. This type of contracts is specified as weak since, unlike for strong contracts, their satisfaction in every context should not be mandatory. For example, if we use the Estimator component for estimating fuel level in the tank of a vehicle, then omitting to display the value would be safer than displaying the wrong value.

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, they are expressed by strong contracts. The corresponding contracts state in which cases the component guarantees that it will not exhibit a failure. We do this by guaranteeing the rule that describes this behaviour, as shown in Table 6.2 for the $\langle A, G \rangle_{Estimator-2}$ contract.

The third type of safety contracts that we translate from FPTC rules are

Table 6.2: The translated contract examples for the Estimator component

Id	Assertions
$\mathbf{A}_{Estimator-1}$:	{sV, p1}.failure within {omission, valueCoarse};
$\mathbf{G}_{Estimator-1}$:	eSV.failure within{omission} AND not eSV.valueCoarse;
$\mathbf{A}_{Estimator-2}$:	-;
$\mathbf{G}_{Estimator-2}$:	sV.valueCoarse, p1.valueCoarse $\rightarrow$ eSV.noFailure;
$\mathbf{B}_{Estimator-3}$:	not sV.omission and not p1.omission;
$\mathbf{H}_{Estimator-3}$:	not eSV.omission;

related to the failures that have been mitigated and do not occur on the output port in any of the specified FPTC rules (e.g., valueCoarse failure for the Estimator component). An example of a such contract is shown in Table 6.2 for the $\langle A, G \rangle_{Estimator-1}$ contract where assumptions are made on the failure modes on the input ports considered by the FPTC rules. The component guarantees that if no other failures occur on the inputs than the ones considered by the FPTC rules, then only the omission failure can occur on the specific output, while the valueCoarse failure will not occur on the output. The guarantee explicitly specifies which failure will not occur on the specific output based on the current FPTC analysis to avoid an implicit interpretation that all failures that do not occur on the output are mitigated by the component. The assumptions for this contract represent the set of failure modes explicitly considered within the FPTC rules for each of the input ports. As mentioned in Section 6.1.2, to use FPTC rules of a component developed in isolation in a particular system, the set of failure modes considered for the component and the system should be the same. Since it is not always reasonable to consider all failure modes for all ports [13], the assumptions of this contract ensure that if a failure mode not considered by the FPTC rules can occur on the corresponding port of the component in the particular system, then such FPTC rules cannot be used until they are updated to take in consideration the missing failure mode.

As shown in the example of translating FPTC rules from Table 6.1 to contracts in Table 6.2, the translation can be performed in the following way for each failure:

- Consider all the rules and the failure modes used for each port and make a strong contract:
  - Identify the list of all the distinct explicitly specified failure modes

for each of the input ports and add them as assumptions connected with AND operator;

- For each of the output ports:
  - Add a guarantee stating the set of failure modes that can occur on the specific output connected with AND operator;
  - Calculate the set difference of the set of considered failure modes on the inputs with respect to the set of failure modes that occur on the output and add the negation of those failure modes for the particular ports as guarantees connected with AND operator;

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

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.

6.2.2 HSFM argumentation pattern instantiation

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 additionally be 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 6.1.3. We adapt the top-level argument to further develop the sub-goals.

Since we can identify the primary and secondary causes from FPTC analysis and the derived contracts, we use the safety contracts to develop 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 composed from the instantiations of the requirement and contract satisfaction assurance argument patterns (Figures 4.3 and 4.4) for each relevant safety contract. If a contract addresses a particular safety requirement, which is related to a particular failure mechanism, then the confidence in satisfaction of such requirement and the contract is argued under the corresponding failure mechanism argument branch. By combining the contract-driven assurance argument patterns with the existing HSFM argument pattern we show how contract-driven assurance can be used in combination with existing assurance strategies.

6.3 Summary

In this chapter we have presented a method that puts the assurance and reuse aware contract-based design and development in the context of COTS architectures where failure logic analysis is usually performed independently of the system. We have used FPTC analysis as it supports compositional failure logic analysis of a system, where its specifications describing failure behaviour of a component can be established in isolation, and not necessarily in the context of a particular system. Using such analysis to derive safety contracts allows us to reap the benefits of assurance and reuse aware contract-based design when moving analysis results established in isolation to a particular system. More specifically, transforming the FPTC specification to contracts allows us to associate assurance information directly with the FPTC specification, which is rather important for establishing confidence in its results. Even more impor-

tantly, this transformation enables utilising the contract variability to identify which FPTC specification is relevant to assure in the final argument, depending on their relation to the corresponding safety requirements.

Since FPTC analysis establishes fault propagation rules between input and output ports of a component, it can be useful for assuring absence of particular failure modes in the component. We have built upon an established argument pattern for assuring absence of hazardous software failure modes and related it with the contract-driven assurance approach. By connecting the proposed contract-driven assurance with existing argumentation strategies, we show that contract-driven assurance can be used as a basis for instantiating other argumentation patterns and that it does not have to be a standalone assurance technique.

Chapter 7

Tool support

In this chapter we present our efforts to provide tool support for the assurance and reuse aware contract based design. The implementation of the tool support is done within the AMASS platform[1] developed as part of the AMASS [74] project. The platform encompasses different tools, but here we focus on the three tools that facilitate system modelling (CHESS), formal verification of assumption guarantee contracts (OCRA), and assurance case modelling (OpenCert). An overview of the three tools is shown in Fig. 7.1. We build upon the synergy of the three tools and implement the contract-driven assurance and reuse methodology by developing new and upgrading the existing plugins within the tools. We extend CHESS to support SEooCMM by adding the possibility to capture information about assurance assets and their relation to the corresponding contracts. With OCRA results back-propagated to the CHESS model, we perform automated weak contract filtering for the component instances. Upon updating the CHESS model, we then automatically instantiate the contract-driven assurance argumentation patterns for each component in the CHESS model. The generated argumentation is stored on a CDO server which can be accessed by any OpenCert argumentation editor connected to the CDO server. In the reminder of the section we detail the implementation (available in the CHESS [2] and OpenCert[3] repositories) of refinement checking with strong and weak contracts as an extension of CHESS and the automatic argument generation as an OpenCert plugin.

[1] https://www.polarsys.org/opencert/
[2] https://git.polarsys.org/c/chess
[3] https://git.polarsys.org/c/opencert

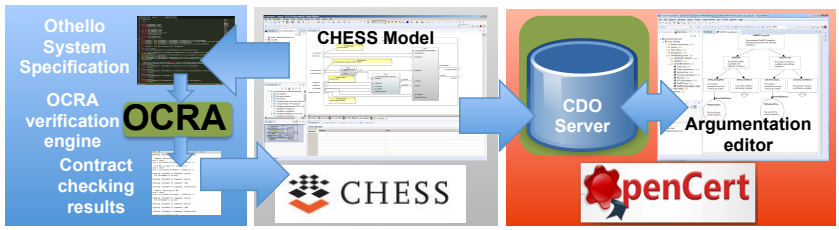

Figure 7.1: The overview of the tool information flow

7.1 Contract-driven reuse support with CHESS and OCRA

As mentioned in Section 5.2, to use a contract checking engine such as OCRA, which does not distinguish between strong and weak contracts, we can either support *weak contract filtering* as a part of reusable component instantiation or *weak contract transformation* to an appropriate format. We extend CHESS so that we can check all the weak contract validity and automatically update the component instance by indicating which weak contracts are valid in the given environment.

To fully support the presented methodology, we have also implemented the second solution that includes all weak contracts in contract refinement checking. The choice of which type of refinement with strong and weak contract to use is up to the user. This is to allow for different possibilities. When the users are manually selecting which weak contracts they want in the given context, then they may have to manually check which of them are relevant for their system. Conversely, when the user selects to perform refinement check with all the weak contracts, then if any of the weak contracts meet the system demands, the refinement will be successful and the weak contracts applicable in the given context will be automatically indicated without the need to manually select them. Our CHESS extensions to support the contract-driven assurance and reuse are hosted in the following CHESS plugins:

- *org.polarsys.chess.contracts.transformations* – contains model to text [75] transformation for generating the .oss file representing the model;

- *org.polarsys.chess.contracts.integration* – contains the interface for communicating with OCRA.

7.1.1 Methodological Guidance

As presented in Chapter 5.2, strong and weak contracts are introduced for out-of-context reasoning and component reuse across a variety of environments. While strong contracts must hold in all environments, the weak ones are environment specific. Prior to performing the refinement check using strong and weak contracts, we create contracts and allocate them to the component types (they represent out-of-context components). At the component type level, we indicate if a contract is strong or weak. When the component type is instantiated in a particular system to a component instance, all the strong, and a subset of weak contracts can be identified as relevant in the particular system in which the component is instantiated. Identifying those relevant weak contracts can be done manually on the component instance level. For example, Figure 7.2 depicts the selected weak contracts for the *Select_Switch_Impl* component instance. The *Contract Refinement Analysis (OCRA)* command transforms the CHESS model into an Othello System Specification (.oss) file readable by OCRA, then it runs the OCRA refinement check and outputs the results. Since the .oss format does not explicitly distinguish between strong and weak contracts, but treats all contracts as strong, the weak contracts need to be accordingly transformed to strong contracts for the .oss format. To perform the refinement check with strong and weak contracts, we first create a *ContractRefinementAnalysisContext* in the *DependabilityAnalysisView* where we can select the platform that should be analysed. To run the refinement check by considering all the contracts the attribute *checkAllWeakContracts* needs to be set to true (Figure 7.3). If the *checkAllWeakContract* attribute is set to false, the refinement check will be performed such that the selected weak contracts on the component instance level will be treated equally as the strong contracts in the generated .oss file. In this case we need to either manually ensure that we do not select contradictory weak contracts, or automatically select the relevant weak contracts by running the *Validate Properties (OCRA)* command. The *Validate Properties (OCRA)* command will check validity of each weak contract assumption and identify which weak contract assumptions are met in the given system. Upon running the weak contract assumption validity check, the contract status is updated accordingly and those contracts are automatically selected in the corresponding component instance. Now it is possible to run the refinement check with the flag *checkAllWeakContracts* set to false without it prompting an inconsistency error. It should be noted that a limitation of the *Validate Property* command is that it can be executed only with a discrete-time specification, hence the usage of continuous variables or operators in the

contracts disables the validity property check.

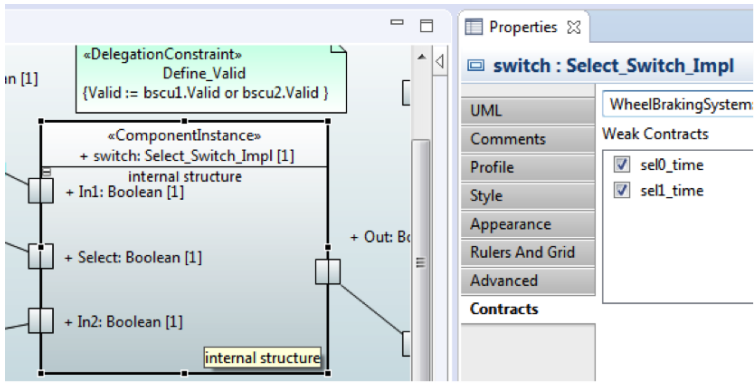

Figure 7.2: Selecting the relevant weak contract for a component instance

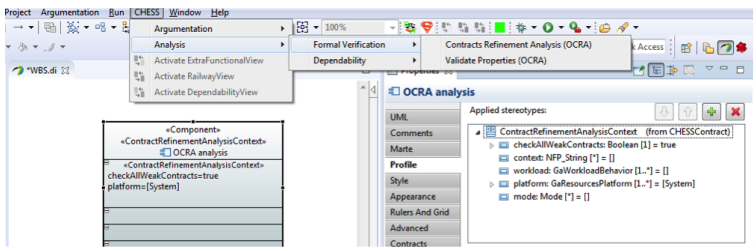

Figure 7.3: Performing the refinement analysis with strong and weak contracts

7.2 Contract-driven assurance support with CHESS and OpenCert

To facilitate automated instantiation of the contract-driven assurance pattern from Section 4.2, we implement the *ArgumentGenerator* plugin[4] within OpenCert. The user is prompted to select both the source CHESS model and the target

[4]org.eclipse.opencert.chess.argumentGenerator

assurance case in the CDO repository. The plugin generates a set of argument-fragments from the source CHESS model and stores them in the corresponding target assurance case in the CDO repository. The ArgumentGenerator assumes that the CHESS model contains contract specifications and that the contract refinement check has been performed such that the status of both strong and weak contracts is updated to indicate if the contract is validated in the given context or not. The argument generation creates an argument-fragment for each component. The connection between different argument-fragments is done through away goals. The resulting argument-fragments can be viewed in the target assurance case by anyone with access to the CDO server from an OpenCert argumentation editor.

7.2.1 Methodological Guidance

The Argument Generator assumes that the CHESS model is enriched with contracts and that contract refinement has been performed such that the contract status is updated to indicate if the contract is validated in the given context or not. Moreover, the Argument Generator assumes that the analyzed model and the refinement check results are stored in the refinement analysis context. The attached screenshots (Figure 7.4 - Figure 7.8) illustrate the usage of the Argument Generator plugin. First, the plugin prompts a selection of the OCRA refinement analysis context (Figure 7.4-Figure 7.5). Then, we need to indicate to which assurance case on the corresponding CDO server the argument-fragments should be stored (Figure 7.6). The argument-generation is performed for each component and for each validated contract (Figure 7.7). The set of argument-fragments for each component can be viewed in the selected assurance case (Figure 7.8).

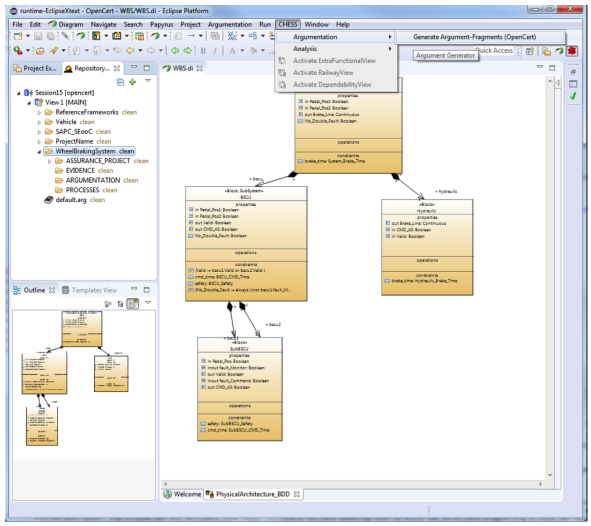

Figure 7.4: Initiating the argument-fragment generation

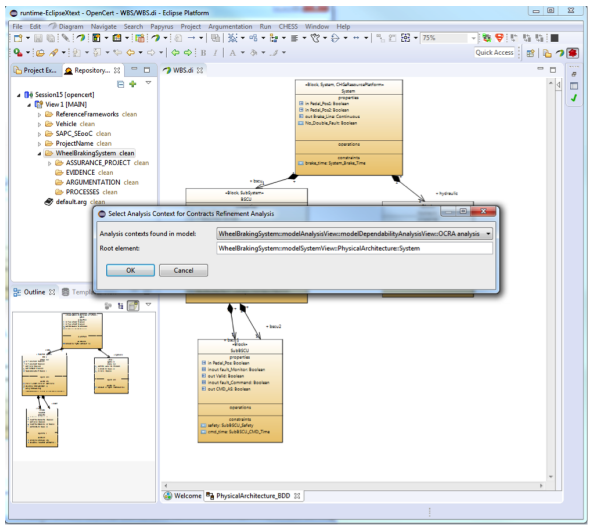

Figure 7.5: Selecting the source analysis context

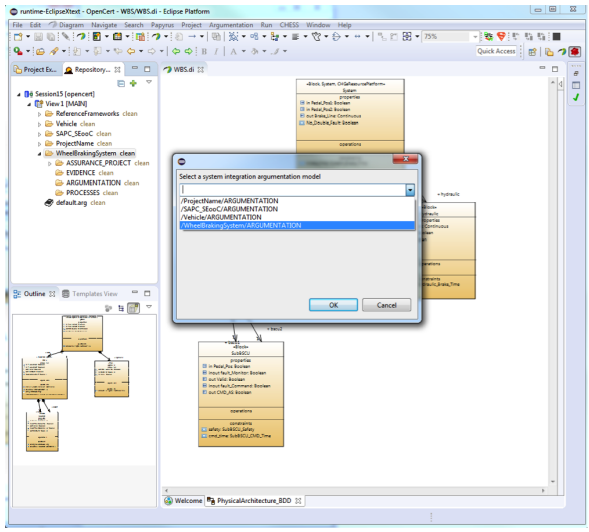

Figure 7.6: Selecting the target assurance case folder on the CDO repository

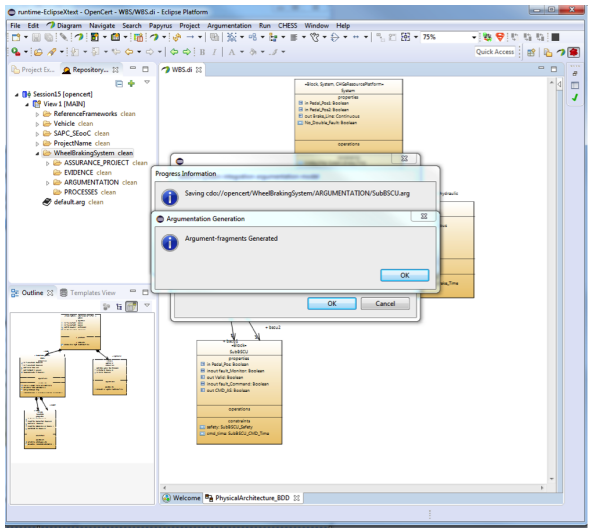

Figure 7.7: Successful generation of argument-fragments for each component

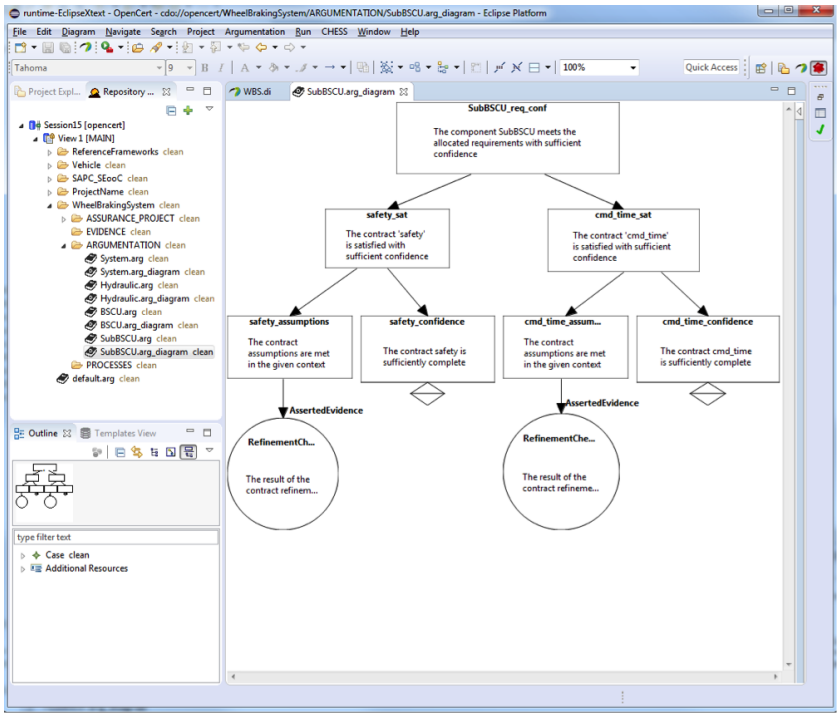

Figure 7.8: An example of a generated argument-fragment

Chapter 8

Validation

In this chapter we present two case studies performed in the context of the Loading Arm Controller Unit system described in Chapter 3. We first describe the case study research method, and then present the two case studies separately, each following the same guidelines.

8.1 Case Study Method

Case study is an empirical method for investigating a contemporary phenomenon in its real-world context [17]. Initially, case studies in software engineering were used for demonstration purposes, as a way to demonstrate an implementation of a particular software technology or concept. At the same time, case studies were already established in social domains as a way to both explore a phenomenon but also to improve it. In general, case studies just as other research methodologies can be used for four different research purposes [17]:

- Exploratory – seeking new insights and generating ideas by studying the phenomenon.
- Descriptive – describing a situation or phenomenon.
- Explanatory – seeking to understand as to why a particular situation or a problem occurs. This also includes confirmatory case studies, that try to test existing theories.
- Improving – trying to improve some part of the studied phenomenon.

Regardless of the purpose of a case study, its crucial part is *the case* [76]. The *case* is the object of study and should be a sufficiently complex component investigated in its natural and real-world context. Unlike other research methodologies, case studies focus on the realism of the studied case. This often implies relinquishing control for the sake of realism, because the realistic situation or phenomenon is often more complex due to the difficult to identify all the variables and their causal relationships. Due to this fact, it is difficult to base case studies on quantitative data analysed using statistics, but rather, the case studies tend to be based on qualitative data that are generally richer and more detailed in description than quantitative data. Hence, the conclusions of a case study will not have the same statistical significance as those for example from an experiment. But unlike an experiment, the case study method allows for studying a more complex phenomenon, where the conclusions of the case study try to link different kinds of evidence and information in order to support a particular statement about the studied phenomenon.

Another consequence of the realism is that the design of the case study is generally flexible. The flexibility of the process of performing the study allows for changing some of the parameters of the study during its execution. While this would invalidate some research methodologies such as experiments, it is a natural occurrence for case studies due to the degree of realism as well as the lack of control and full awareness of the studied case and its environment.

We present the case studies using the following structure:

- *Case study design* – contains the objective of the case study and the research questions we aim to answer. The design section also addresses the relevance of the case itself.

- *The case definition* – describes the case under study with all of its technical details needed for data collection and discussions.

- *Data collection* – represents a set of techniques applied to the case such that they generate data sufficient to build the conclusions of the case study.

- *Discussion* – presents positive and negative conclusions regarding the case. It is much like an assurance case argument, which chains different information and evidence to support statements about the studied case.

- *Validity* – addresses different validity threats that may undermine the conclusions from the case study and how these thereafter are mitigated.

8.2 Case Study 1

In this section we first present the case study design in Section 8.2.1, and then introduce the particular case of study in Section 8.2.2. In Section 8.2.3 we present the SEooC definition and development, and then in Section 8.2.4, we describe the LAAP integration within two different products of a wheel-loader product-line. In Section 8.2.6 we provide a discussion and then examine the case study validity in Section 8.2.7.

8.2.1 Case Study Design

The objective of this case study is to apply the contract-aware SEooC development and reuse, and evaluate the feasibility of fine-grained reuse of a component and its accompanying assurance information in two contexts that have different safety requirements. More specifically, we perform an explanatory case study to answer the following research question:

- **RQ1**: *Can the contract-aware SEooC development and assurance method be used to identify which assurance information is relevant for safety assurance in a given context based on the contract specifications?*

We consider a case where a reusable component is developed out-of-context as a SEooC, following the contract-aware SEooC development and reuse approach presented in Section 5.3, and reused in two different products. In particular, we consider that the Loading Arm Automatic Positioning (LAAP) component as a part of the Loading Arm Control Unit (LACU) described in Section 3, is developed as a SEooC. The two different products in which the LAAP is reused impose different safety requirements on the LAAP. Hence, the contract-aware SEooC development should take in account the different safety implications, and during the integration of the component, it should facilitate filtering of the relevant assurance information and finally result in context-specific safety assurance arguments in both cases.

We have selected this scenario for our case study based on discussions with our industrial partners. It is not enough to simply accompany a reusable component with all the assurance information relevant for all the different contexts. There is also a need to identify the relevant subset of that information in a particular context. We have defined LAAP as the reusable component in this scenario together with our industrial partners. Starting from the abstract system models and different product specifications, we have defined the different

hazards LAAP can contribute to, and safety requirements relevant in the different products. Although we did not have access to the actual implementation of the two different products, we have used the relevant system specifications as basis for analysing the different safety implications. We have then been able to use the contract-aware SEooC development and assurance method even with the given specifications of the studied systems.

8.2.2 The Case

The SEooC we develop is the Lifting Arm Automatic Positioning (LAAP) component used within loading arm controllers of various wheel-loader types. LAAP is part of the LACU described in Section 3. Here, we add some additional information regarding LACU that are specific for this case study.

The two wheel-loaders in which we reuse the developed SEooC belong to the same product line. The first wheel-loader 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 wheel-loader is a Small Wheel-loader (SWL) used for less intensive tasks and often outside of construction sites. It is more compact than the GWL and it has two times faster lifting arm raise time, i.e., around 5s.

8.2.3 SEooC definition and development

As discussed in Section 5.3.2, the development of a SEooC starts by making assumptions on the *item* in terms of ISO 26262, 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 Figure 8.1. 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, a recorded position to which the arm should be moved and a sensor that tracks ground speed of the vehicle. 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

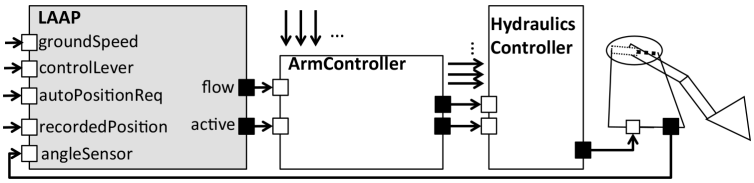

Figure 8.1: The assumed structure of the lifting arm unit context

its relation to possible hazards. We identified contributions of LAAP to two possible vehicle-level hazards, as described in Section 3.2.2:

- *(H1) unintended movement of the lifting arm.*
- *(H2) high-pressure hydraulic leakage.*

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.

One way in which LAAP can contribute to this situation is when the LAAP starts operating but due to a 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 dangerous to the driver, other participants in traffic and bystanders present in the area. Table 8.1 presents the software safety requirements derived from the assumed functional safety concept that address the possible hazardous events related to both hazards.

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 8.2. We specify the contracts using the OCRA compliant Othello constraints syntax introduced in Section 2.5.1. 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* assumes the correct input value range for the *controlLever* signal and guarantees that the safe state shall be applied when

Table 8.1: SW Safety Requirements

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

inputs other than *controlLever* are out of bounds. Moreover, since the LAAP component can be active only when the control lever is inactive (i.e., when *controlLever* is 0), the *LAAP-2* contract also guarantees that the safe state shall be applied when *controlLever* is different from 0.

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 (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. 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 is presented in Table 8.3. The contract *LAAP-4* is not fully addressing the requirement *SWSR2*, since it only establishes that the accuracy of the flow command is dependent on the accuracy of the angle sensor and the recorded posi-

Table 8.2: LAAP Safety Contracts

Id	Assertions
$\mathbf{A}_{LAAP-1}$:	**always** (*groundSpeedLimit* >= 0 and *groundSpeedLimit* <= 20);
$\mathbf{G}_{LAAP-1}$:	**always** ((*groundSpeed* > *groundSpeedLimit*) **implies** (not *active* and *flow* = 0));
$\mathbf{A}_{LAAP-2}$:	**always** (*controlLever* >= -1 and *controlLever* <= 1);
$\mathbf{G}_{LAAP-2}$:	**always** ((*groundSpeed* < 0 or *groundSpeed* > 200 or *angleSensor* < 0 or *angleSensor* > 3 or *controlLever* != 0 or *recordedPosition* < 0 or *recordedPosition* > 3) **implies** (not *active* and *flow* = 0));
$\mathbf{A}_{LAAP-3}$:	**always** (*watchdogTimerInterval* > *raiseTime* and *watchdogTimerInterval* < 1.2**raiseTime* and *raiseTime* > 0);
$\mathbf{G}_{LAAP-3}$:	**always** (((*active* or *flow* != 0) **implies** watchdogTimerActive) and ((*LAAP-OperationalTime* > *watchdogTimerInterval*) **implies** (not *active* and *flow* = 0 and watchdogTimerReset)));
$\mathbf{B}_{LAAP-4}$:	**always** (not *fault_angleSensor_value* and not *fault_recordedPosition_value*);
$\mathbf{H}_{LAAP-4}$:	**always** (not *fault_flow_value*);
$\mathbf{B}_{LAAP-5}$:	**always** (not *fault_autoPositionReq_comission* and not *fault_controlLever_omission*);
$\mathbf{H}_{LAAP-5}$:	**always** (not *fault_flow_comission* and not *fault_active_comission*);
$\mathbf{B}_{LAAP-6}$:	**always** (*angleSensorAccuracy* = 0.02 and actuationDeviation = 0.01 and not *fault_recordedPosition_value*);
$\mathbf{H}_{LAAP-6}$:	**always** (*flowAccuracy* = 0.01 **implies** ((*stopPosition* - *recordedPosition*) >= -0.04 and (*stopPosition* - *recordedPosition*) <= 0.04));

tion 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 5.3, the SW safety requirements addressed by the safety contracts are supported with evidence through the connection of the contracts and the supporting evidence. The statements that provide clarifications of the contracts and the supporting evidence attached during the Safety Contract Production phase are shown in Table 8.4. The context statements are denoted with *LAAP-x_Cy* and evidence with *LAAP-x_Ey*., where *x* is the number of the related contract and *y* the number of the evidence/context statement.

All the information related to the safety requirements, contracts, evidence,

Table 8.3: SW Safety Requirements and safety contracts mapping

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

as well as assurance information are captured in the CHESS model. We present the information here in the tabular form for the sake of readability. The evidence items are specified in OpenCert and referenced in CHESS in relation to the contracts. The tool usage is described in Chapter 7.

8.2.4 SEooC Integration

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 relevant for 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 are not ensured for the *LAAP-4* contract, and the assumption on actuation accuracy for the *LAAP-6* contract does not hold.

In contrast to the GWL, the SWL is used in less controlled environments for tasks that usually require precision where 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 assumptions of the contract *LAAP-6*, while the assumption of the contract *LAAP-4* is not satisfied in the SWL system as it would be too expensive to achieve it.

Since the strong contract *LAAP-1* requires *groundSpeedLimit* to be below

Table 8.4: The context statements and evidence of the LAAP safety contracts

Id	Statement
LAAP-1_C1:	Assuming the ground speed limit is set below 20km/h, LAAP shall be disabled when the ground speed of the vehicle exceeds the ground speed limit.
LAAP-1_E1	*name*: Unit testing results;
LAAP-2_C1:	Assuming that the control lever signal is not faulty, LAAP shall be disabled when any of the other inputs is out of bounds.
LAAP-2_E1	*name*: Unit testing results;
LAAP-3_C1:	The watchdog timer interval is set relative to the maximum raise time of the lifting arm such that it resets if the arm does not complete its movement within the specified time.
LAAP-3_E1	*name*: Watchdog inspection report
LAAP-3_E2	*name*: Unit testing results;
LAAP-4_C1:	In order to avoid the flow command value failure, the environment of LAAP should guarantee that the angle sensor and recorded position signals do not exhibit value failures;
LAAP-4_E1	*name*: LAAP FPTC analysis report *supporting argument*: FPTC_analysis_conf;
LAAP-5_C1:	In order to mitigate inadvertent commands sent from LAAP, the environment should ensure that commission of the auto positioning request and omission of the control lever signals do not occur.
LAAP-5_E1	*name*: LAAP FPTC analysis report *supporting argument*: FPTC_analysis_conf;
LAAP-6_C1:	Given the accuracy of the angle sensor and the actuation, and assuming that the recorded position is not faulty, the resulting LAAP flow command accuracy ensures the accuracy of the stop position compared to the recorded position.
LAAP-6_E1	*name*: LAAP FPTC analysis report *supporting argument*: FPTC_analysis_conf;
LAAP-6_E2	*name*: Unit testing results;

20 km/h in every vehicle, both products must set the appropriate values. In the GWL the limit is 20 km/h, since the arm moves 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.

8.2.5 Generated Safety Arguments

Figure 8.2 shows the top level goals of the LAAP safety argument for the two systems. The argument generation process from CHESS to OpenCert gives us the building blocks in terms of instantiated requirement and contract satisfaction patterns. We use those argument-fragments in building the system specific arguments. In the top-level argument shown in Figure 8.2, we first present the strong contracts, as those are the ones we are certain will always have to be satisfied. Hence, this part of the argument would be the same for this component across different systems. For the sake of brevity, the goal related to the satisfaction of the *LAAP-2* contract is left undeveloped.

The argument is further decomposed to argue over satisfaction of each allocated safety requirement. There we use the instantiations of the requirement satisfaction pattern. As discussed in Section 8.2.4, some of the contracts are not satisfied in the GWL and at the same time some of the requirements are discarded as quality management, hence not included in the LAAP safety argument in context of GWL. SWSR2 and SWSR4 are not included in the GWL safety argument (Figure 8.4), while for the SWL, all six requirements are included in the corresponding argument (Figure 8.3).

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). We have structured the LAAP argument by focusing on the strong contracts separately from the weak contracts to demonstrate the use of the contract variability enabled by the strong and weak contracts. While the top-level argument-fragment, in particular the *G2* goal, would be the same when we reuse LAAP in different systems, the requirement satisfaction argument-fragment is specific for each system.

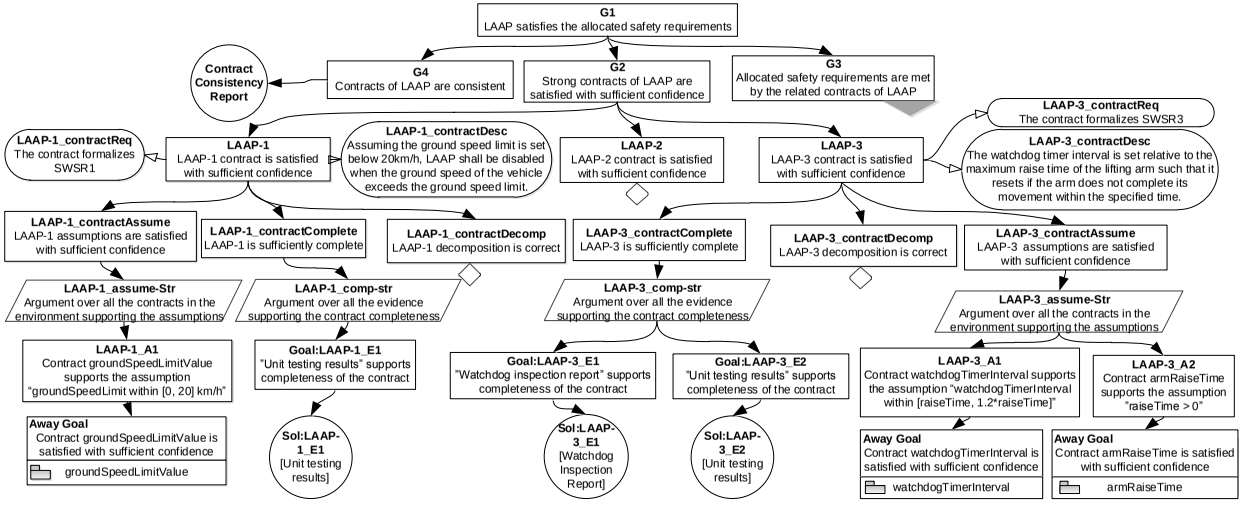

Figure 8.2: Top goals of the LAAP safety argument with the strong contracts argument-fragment developed (the same for both the GWL and SWL)

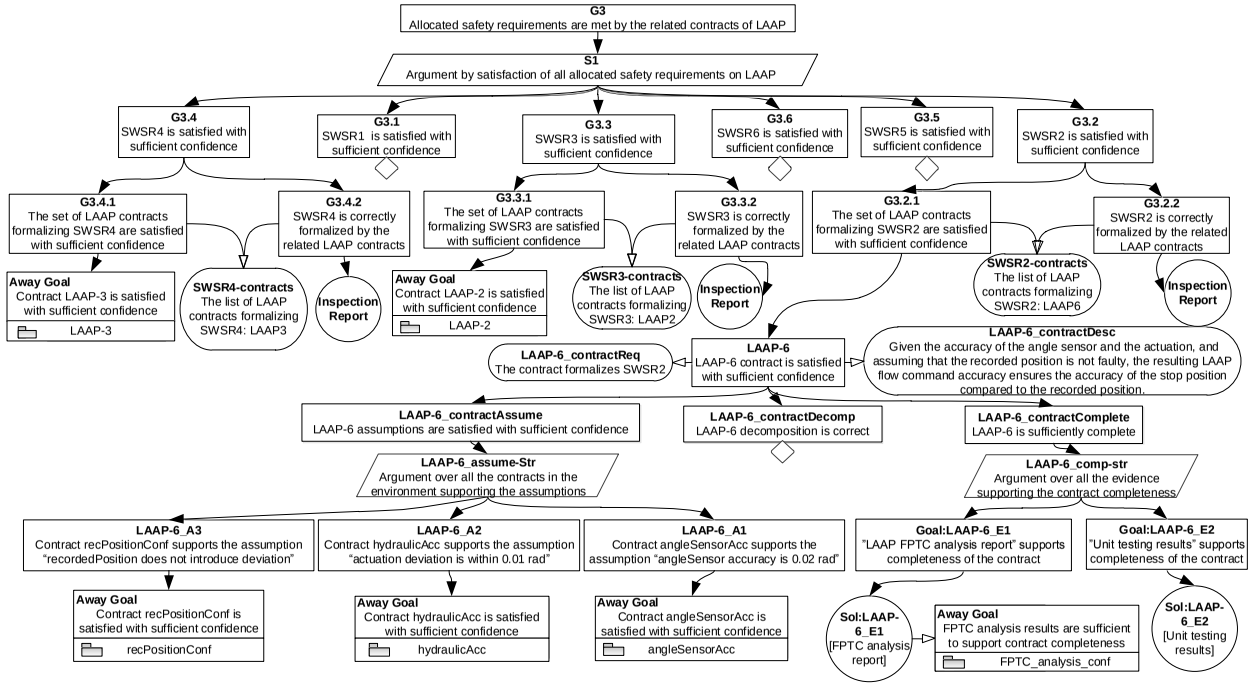

Figure 8.3: Safety argument-fragment for the safety requirements allocated on the LAAP in context of SWL

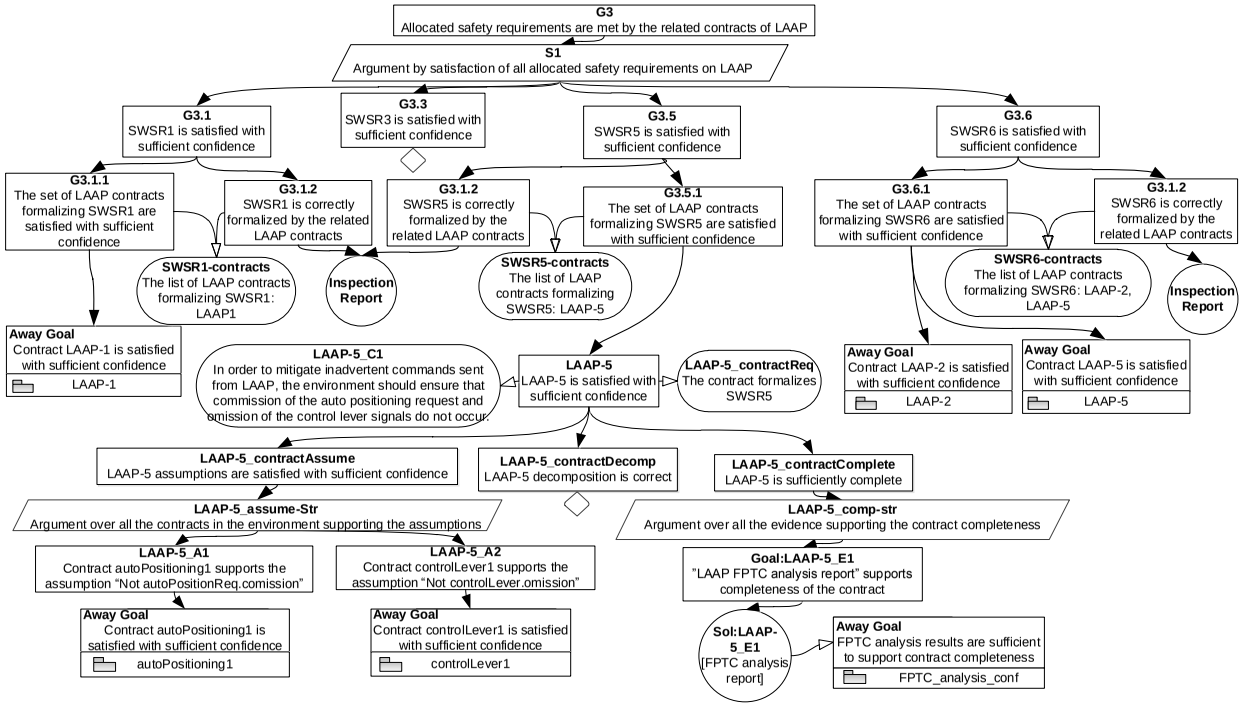

Figure 8.4: Safety argument-fragment for the safety requirements allocated on the LAAP in context of GWL

8.2.6 Discussion

As described in Section 2.3, ISO 26262 requires certain information to be gathered during the concept phase of the development of an item in order to define the functional safety concept for the item. The standard mandates that the functional safety concept should be considered and refined at the implementation level of the software and hardware elements of the item. In the case of SEooC, the information related to the functional safety concept 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. In this case study we have used the assurance and reuse aware contract-based design as a way to assume, consider and validate this information together with additional benefits of automated assurance and fine-grained reuse. 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 safety contracts assumptions are satisfied in the particular system.

As presented in Section 8.2.4, 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. This is supported by the assurance and reuse aware contract-based design, where contract checking is used to identify the relevant information in a particular context. We have presented the context-specific argument-fragments for the different systems to show that the contract-aware SEooC development and reuse can assist in identifying the relevant safety assurance information, and in that way we have addressed the main research question of the case study (RQ1).

8.2.7 Validity

When designing this case study, our main focus was on specifying a realistic reuse scenario between at least two different systems. We have done this with our industrial partners, based on their experience on working with similar systems. Once we have identified the reusable component, we needed some

system specifications and context descriptions of the two systems, to be able to identify the hazards and the corresponding safety requirements for both systems. We have derived detailed system specifications from system descriptions and high-level functional specification of the system. Having the system information enabled us to check the system assumptions made during the SEooC reuse, which helped us identify the relevant information for each system using contract checking. The results depend on the realism of both the defined reusable component and the system specifications. Although we tried to use the industrial experience when dealing with both steps to make the case as realistic as possible, in the end the case is always simplified to a degree that may have influenced the case study conclusions.

Designing a SEooC in a realistic scenario would mean making assumptions about the different target systems in which the SEooC may be reused. In our case, the realism here was influenced by the fact that we were also at the same time defining the two target systems, which influenced our process of making the system assumptions while designing the SEooC. In reality, the assumptions would not necessarily match with the in-context requirements [77], which could require some additional work in aligning the SEooC assumptions and system specification.

8.3 Case Study 2

In this section we document the second case study by following the same structure presented in Section 8.1. We first present the case study design in Section 8.3.1, and then introduce the particular case of study in Section 8.3.2. In Section 8.3.3 we apply CHESS-FLA/FPTC analysis on a reusable component and use the translation steps from Section 6.2.1 to translate the FPTC analysis results to the contracts. Next, we finalise the CHESS-FLA/FPTC analysis in context of a specific system and present the system level contracts in Section 8.3.4. We present the generated argument-fragment in Section 8.3.5. In Section 8.3.6 we provide a discussion and then examine the case study validity in Section 8.3.7.

8.3.1 Case Study Design

The objective of our case study is to apply FLAR2SAF on a real-world case commonly found in industry and evaluate the feasibility of reuse and generation of safety artefacts related to FPTC analysis. More specifically, we conduct

an explanatory case study to answer the following research questions:

- **RQ2.1**: *Can FPTC analysis be performed if inputs of all components under analysis do not consider the same set of failures?*
- **RQ2.2**: *Is safe reuse of FPTC-related safety artefacts possible when the set of failures considered in the FPTC analysis of the reusable component does not match the set of failures from the FPTC analysis of the system in which the component is reused?*

We consider a case where a component is developed independently of a single system and then reused in a system that is part of a family of products. More specifically, a functionality of a Loading Arm Control Unit (LACU) is reused within a wheel-loader product-line. The functionality being reused is an independently developed Loading Arm Automatic Positioning (LAAP) component that supports FLAR2SAF.

We have selected this particular case based on industrial needs. Companies that develop ranges of products with similar functionalities often face a similar scenario: they reuse components in different products, but not the accompanying safety artefacts. In cooperation with our industrial partners, we have defined the case scenario and developed it further based on an abstracted model of the system. Although we did not have access to the actual implementation of the system, we have been able to apply FLAR2SAF since we had sufficient knowledge of the failure behaviour of the system.

8.3.2 The Case

In this case study we focus on a particular case of the LACU system described in Section 3. The aim is on reuse of LAAP within a Small Wheel-loader (SWL), which is often used for tasks that require high precision of the arm movement. Moreover, SWL is not used only in construction sites, but also for public service in areas with pedestrians.

The software architecture of LACU modelled in CHESS is shown in Figure 8.5. The hazard analysis of the loading arm has identified a vehicle level hazard *H1: unintended movement of the lifting arm*, which can be dangerous in different operational situations in which SWL is used. Angle sensor value failure is identified as a contributor to the hazard H1. As one of the safety measures implemented to mitigate this hazard, the angle sensor is duplicated and monitored in software to protect against value failures. The values of both angle sensors are compared by the monitor component both to each other, and

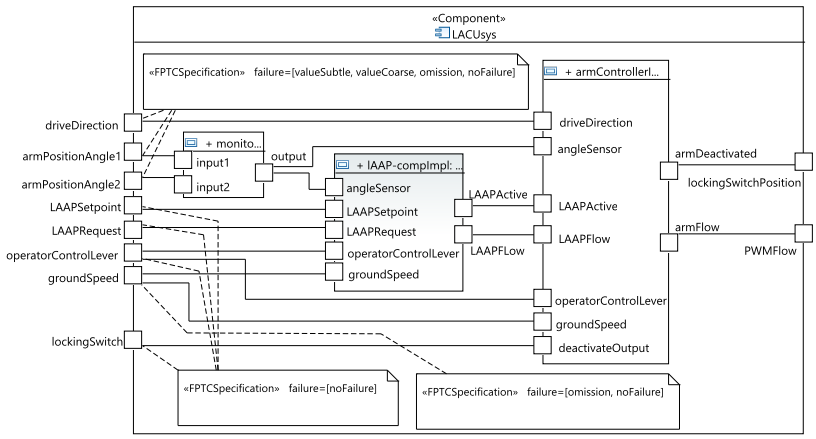

Figure 8.5: LACU model in CHESS

to earlier sensor data to detect value anomalies. While the two sensors can have different accuracy and the sensed values can slightly differ, we do not consider such minor deviations as failures within our FPTC analysis. Furthermore, an error-detecting code is used to detect any accidental changes to the stored variables, such as the predefined position to which the arm should be moved.

Unlike the Monitor component, LAAP is developed out-of-context, with FPTC analysis performed and the resulting failure behaviour captured in safety contracts. The LAAP component enriched with contracts and the accompanying evidence is reused in the context of SWL. In the next section we will focus on the FPTC analysis of LAAP and present its contracts and the accompanying evidence.

8.3.3 LAAP Failure Logic Analysis

The LAAP component is highlighted in the LACU architecture in Figure 8.5. LAAP is activated with the *LAAPRequest* signal issued by the operator. Provided that the *angleSensor*, *groundSpeed*, and *operatorControlLever* are within the specified boundaries, *LAAPActive* is set to true, and the calculated arm movement command is provided through the *LAAPFlow* output. In the reminder of this section we focus on the FPTC analysis part of the LAAP out-of-

«Component»
LAAP-compImpl
«FPTC»
fptc=groundSpeed.omission, operatorControlLever.wildcard, angleSensor.wildcard, LAAPRequest.wildcard, LAAPSetpoint.wildcard-> LAAPFlow.noFailure, LAAPActive.noFailure;
groundSpeed.wildcard, operatorControlLever.omission, angleSensor.wildcard, LAAPRequest.wildcard, LAAPSetpoint.wildcard-> LAAPFlow.noFailure, LAAPActive.noFailure;
groundSpeed.wildcard, operatorControlLever.wildcard, angleSensor.wildcard, LAAPRequest.omission, LAAPSetpoint.wildcard-> LAAPFlow.noFailure, LAAPActive.noFailure;
groundSpeed.wildcard, operatorControlLever.wildcard, angleSensor.omission, LAAPRequest.wildcard, LAAPSetpoint.wildcard-> LAAPFlow.noFailure, LAAPActive.noFailure;
groundSpeed.wildcard, operatorControlLever.wildcard, angleSensor.wildcard, LAAPRequest.wildcard, LAAPSetpoint.omission-> LAAPFlow.noFailure, LAAPActive.noFailure;
groundSpeed.noFailure, operatorControlLever.noFailure, angleSensor.noFailure, LAAPRequest.noFailure, LAAPSetpoint.noFailure-> LAAPFlow.noFailure, LAAPActive.noFailure;

groundSpeed.noFailure, operatorControlLever.noFailure, angleSensor.valueCoarse, LAAPRequest.noFailure, LAAPSetpoint.noFailure -> LAAPFlow.valueCoarse, LAAPActive.noFailure;
groundSpeed.noFailure, operatorControlLever.noFailure, angleSensor.noFailure, LAAPRequest.noFailure, LAAPSetpoint.valueCoarse -> LAAPFlow.valueCoarse, LAAPActive.noFailure;
groundSpeed.noFailure, operatorControlLever.noFailure, angleSensor.valueCoarse, LAAPRequest.noFailure, LAAPSetpoint.valueCoarse -> LAAPFlow.valueCoarse, LAAPActive.noFailure;

groundSpeed.noFailure, operatorControlLever.noFailure, angleSensor.valueSubtle, LAAPRequest.noFailure, LAAPSetpoint.noFailure -> LAAPFlow.valueSubtle, LAAPActive.noFailure;
groundSpeed.noFailure, operatorControlLever.noFailure, angleSensor.noFailure, LAAPRequest.noFailure, LAAPSetpoint.valueSubtle -> LAAPFlow.valueSubtle, LAAPActive.noFailure;
groundSpeed.noFailure, operatorControlLever.noFailure, angleSensor.valueSubtle, LAAPRequest.noFailure, LAAPSetpoint.valueSubtle -> LAAPFlow.valueSubtle, LAAPActive.noFailure;

groundSpeed.valueSubtle, operatorControlLever.wildcard, angleSensor.wildcard, LAAPRequest.wildcard, LAAPSetpoint.wildcard-> LAAPFlow.commission, LAAPActive.commission;
groundSpeed.valueCoarse, operatorControlLever.wildcard, angleSensor.wildcard, LAAPRequest.wildcard, LAAPSetpoint.wildcard-> LAAPFlow.commission, LAAPActive.commission;
groundSpeed.wildcard, operatorControlLever.valueCoarse, angleSensor.wildcard, LAAPRequest.wildcard, LAAPSetpoint.wildcard-> LAAPFlow.commission, LAAPActive.commission;
groundSpeed.wildcard, operatorControlLever.valueSubtle, angleSensor.wildcard, LAAPRequest.wildcard, LAAPSetpoint.wildcard-> LAAPFlow.commission, LAAPActive.commission;
groundSpeed.wildcard, operatorControlLever.wildcard, angleSensor.wildcard, LAAPRequest.commission, LAAPSetpoint.wildcard-> LAAPFlow.commission, LAAPActive.commission;

Figure 8.6: A subset of LAAP FPTC rules

context development covered in Section 8.2, and detail the translated contracts.

The FPTC rules representing the LAAP failure behaviour are shown in Figure 8.6. The first set of rules describes that the component does not return a failure in case it detects that any of the input values is omitted. Moreover, the component is not a source of failures, hence if there are no failures on the inputs, there will be no failures on the outputs of the component. The second and the third set of rules indicate that valueCoarse/valueSubtle failures of the LAAPFlow command can occur when either angleSensor, LAAPSetpoint, or both exhibit the corresponding valueCoarse/valueSubtle failure. Finally, the last set of FPTC rules describes when the component exhibits commission failures on both of its output ports. Since whenever LAAPActive exhibits com-

Table 8.5: A subset of the translated LAAP strong contracts with the associated evidence

Id	Assertions and support statements
$\mathbf{A}_{LAAP-S1}$:	**always** (*fault_groundSpeed* in {omission, valueSubtle, valueCoarse} and *fault_operatorControlLever* in {omission, valueSubtle, valueCoarse} and *fault_angleSensor* in {omission, valueSubtle, valueCoarse} and *fault_LAAPSetpoint* in {omission, valueSubtle, valueCoarse} and *fault_LAAPRequest* in {omission, commission});
$\mathbf{G}_{LAAP-S1}$:	**always** (*fault_LAAPFlow* in {valueSubtle, valueCoarse, commission} and *fault_LAAPActive* in {commission} and not *fault_LAAPFlow_omission* and not *fault_LAAPActive_omission*;
$\mathbf{C}_{LAAP-S1}$:	For LAAP not to exhibit omission and the FPTC analysis to be reusable with other FPTC specifications, only the specified failure modes can occur on the corresponding input ports.
$\mathbf{E}_{LAAP-S1}$:	*name*: LAAP FPTC analysis report *supporting argument*: FPTC_rules_conf;
$\mathbf{A}_{LAAP-S2}$:	-;
$\mathbf{G}_{LAAP-S2}$:	**always** (*fault_groundSpeed_omission* **implies** (*fault_LAAPFlow_noFailure* and *fault_LAAPActive_noFailure*));
$\mathbf{C}_{LAAP-S2}$:	If ground speed signal is unavailable, the LAAP is disabled to prevent value or commission failures from propagating.
$\mathbf{E}_{LAAP-S2}$:	*name*: LAAP FPTC analysis report *supporting argument*: FPTC_rules_conf; --- *name*: Unit testing results *supporting argument*: Unit_test_conf;

mission, the command LAAPFlow is calculated and also provided when not supposed to, hence the commission of both of the ports is handled jointly. The commission of the two outputs occurs when either the groundSpeed sensor or operatorControlLever exhibit value failures, or when the LAAPRequest command is issued inadvertently. For example, the LAAP component has a built in mechanism to deactivate itself if the operator control lever is active. In this case, an incorrect control lever value can postpone deactivation of the loading arm which results in both signals LAAPActive and LAAPFlow being issued when not supposed to.

From the LAAP FPTC rules we translate the three types of contracts detailed in Section 6.2.1. We specify the contracts using the OCRA compliant Othello constraints syntax introduced in Section 2.5.1. The translated strong contracts are shown in Table 8.5. Since the FPTC rules do not consider all possible failures on its inputs – only those deemed feasible or relevant – the strong contract is used to ensure that the component can be used even though it does not consider all possible failures on its inputs. To achieve this, the strong contract *LAAP-S1* imposes restrictions on the environment of the component by making assumptions that the component can receive on its input ports only those failures considered within the FPTC analysis for this component. More specifically, the component considers omission and commission on LAAPRequest, and omission, valueSubtle and valueCoarse on other input ports. *LAAP-S1* then indicates that the component guarantees that it will not exhibit omission failures, while it can exhibit value and commission failures. If these strong assumptions are not satisfied, then the LAAP FPTC analysis and the translated contracts should be revisited. The second strong contract *LAAP-S2* is an example of a contract where the FPTC rule is guaranteed and its validity is supported by different types of evidence.

The FPTC rules that indicate when valueCoarse, valueSubtle, and commission failures occur are translated to the weak contracts shown in Table 8.6. The FPTC rules about the valueCoarse failure of the LAAPFlow port combined are translated to the contract *LAAP-W1*. The contract states that for the LAAP component not to exhibit valueCoarse failure on the LAAPFlow port, the environment in which the component is used should ensure that the angleSensor and the LAAPSetpoint values do not exhibit coarse value failures. Similarly, the contract *LAAP-W2* states that for the LAAP component not to exhibit valueSubtle failure, the environment should ensure that the angleSensor and the LAAPSetpoint values do not exhibit subtle value failures. Finally, the third contract *LAAP-W3* indicates that to prevent commission of both of the outputs, there should be no value failures on groundSpeed and operatorControlLever

Table 8.6: The translated LAAP weak contracts with the associated evidence

Id	Assertions and support statements
$\mathbf{B}_{LAAP-W1}$:	**always** (not *fault_angleSensor_valueCoarse* and not *fault_LAAPSetpoint_valueCoarse*);
$\mathbf{H}_{LAAP-W1}$:	**always** (not *fault_LAAPFlow_valueCoarse*);
$\mathbf{C}_{LAAP-W1}$:	For LAAPFlow not to exhibit great value failure, neither angleSensor nor LAAPSetpoint should fail with valueCoarse failure mode.
$\mathbf{E}_{LAAP-W1}$:	*name*: LAAP FPTC analysis report *supporting argument*: FPTC_rules_conf;
$\mathbf{B}_{LAAP-W2}$:	**always** (not *fault_angleSensor_valueSubtle* and not *fault_LAAPSetpoint_valueSubtle*);
$\mathbf{H}_{LAAP-W2}$:	**always** (not *fault_LAAPFlow_valueSubtle*);
$\mathbf{C}_{LAAP-W2}$:	For LAAPFlow not to exhibit subtle value failure, neither angleSensor nor LAAPSetpoint should fail with valueSubtle failure mode.;
$\mathbf{E}_{LAAP-W2}$:	*name*: LAAP FPTC analysis report *supporting argument*: FPTC_rules_conf;
$\mathbf{B}_{LAAP-W3}$:	**always** (not *fault_groundSpeed_valueSubtle* and not *fault_groundSpeed_valueCoarse* and not *fault_operatorControlLever_valueSubtle* and not *fault_operatorControlLever_valueCoarse* and not *fault_LAAPRequest_commission*);
$\mathbf{H}_{LAAP-W3}$:	**always** (not *fault_LAAPFlow_commission* and not *fault_LAAPActive_commission*);
$\mathbf{C}_{LAAP-W3}$:	For LAAP not to move the arm inadvertently, the groundSpeed and operatorControlLever signals should not exhibit value failures and the LAAPRequest should not be issued when not supposed to.
$\mathbf{E}_{LAAP-W3}$:	*name*: LAAP FPTC analysis report *supporting argument*: FPTC_rules_conf;

ports, as well as no commission failure on the LAAPRequest port. All three contracts are supported by the FPTC analysis report from which the contracts are derived. Moreover, an additional argument is attached to support the confidence in the specified FPTC rules.

8.3.4 LACU Failure Logic Analysis

As mentioned in Section 8.3.2, the LACU hazard analysis indicates that the value failures of the angle sensor can lead to the hazard H1. Hence, when selecting the subcomponents for this system, their failure behaviour related to the value failures needs to be investigated to ensure that value failures are contained. The contracts derived from the FPTC rules show which conditions need to be satisfied for a particular component not to exhibit such failures, e.g., valueCoarse failure. To be able to reuse the LAAP component in the context of LACU and perform FPTC analysis, the strong contract of LAAP needs to be satisfied, i.e., *there should be no failures occurring on the inputs of LAAP other than those specified in the assumptions of the* LAAP-S1 *contract*. Although LAAP does not consider all possible failures on its input, the FPTC analysis can still be performed and its results are valid in the systems that satisfy such strong contract assumption. For example, the LAAP FPTC rules do not consider value failures of the LAAPRequest port. As long as the system provides guarantees that the received failures on LAAPRequest can only be omission or commission, the analysis can be performed. If the component is reused in a system that allows LAAPRequest to exhibit value failures, then the corresponding FPTC rules of LAAP need to be updated to examine the consequences on the output ports.

Since the strong assumptions are satisfied, we then examine the value failure behaviour of LAAP. We can identify from the contracts *LAAP-W1* and *LAAP-W2* that LACU should ensure that angleSensor and LAAPSetpoint values should not be erroneous for the LAAP component not to exhibit the valueCoarse and valueSubtle failures. As mentioned in Section 8.3.2, the software monitor and the error-detecting code have been implemented to ensure that the contracts *LAAP-W1* and *LAAP-W2* are satisfied.

The Monitor FPTC rules are shown in Figure 8.7. If either value or occurrence error is detected on both inputs, the output is omitted. For a value to be provided to the other components at least one of the inputs should not exhibit a failure. The Monitor output is provided to both the LAAP and ArmController components. Unlike Monitor, the components LAAP and ArmController simply propagate value failures received on their inputs, while they guarantee that

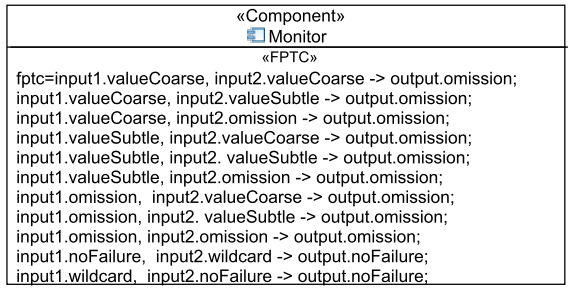

Figure 8.7: The set of Monitor FPTC rules

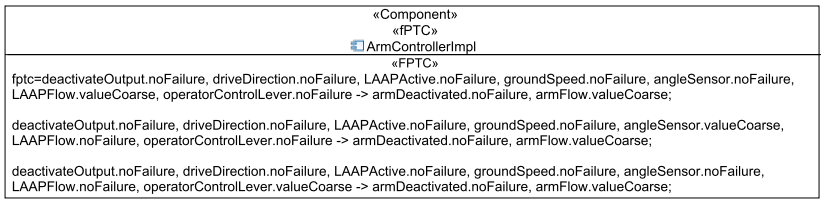

Figure 8.8: A subset of the ArmController FPTC rules

they are not sources of such failures. A subset of the ArmController FPTC rules related to the valueCoarse failure of the armFlow command is shown in Figure 8.8.

To perform the FPTC analysis on the LACU modelled in the CHESS-toolset, FPTC specifications on the input ports of LACU need to be specified. These specifications indicate which failures can occur on the input ports. As can be seen in Figure 8.5, noFailure is specified for most of the inputs to indicate that failures on those ports are handled outside of LACU itself. Conversely, value and occurrence failures are examined for the angle sensor and ground speed ports as they are handled by the LACU component.

Based on the FPTC rules for the subcomponents and the FPTC specifications on the input ports, the FPTC analysis of LACU indicates that on the PWMflow output value and occurrence failures do not occur. While omission is handled within the component, absence of commission depends on the component environment. For commission not to occur, the environment needs to fulfil certain assumptions (Table 8.8), such as locking switch should not ex-

Table 8.7: A subset of the translated LACU strong contracts with the associated evidence

Id	Assertions and support statements
$\mathbf{A}_{LACU-S1}$:	**always** (*fault_lockingSwitch_noFailure* and *fault_driveDirection* in {omission, commission, valueSubtle, valueCoarse} and *fault_groundSpeed* in {omission, commission, valueSubtle, valueCoarse} and *fault_armPositionAngle1* in {omission, valueSubtle, valueCoarse} and *fault_armPositionAngle2* in {omission, valueSubtle, valueCoarse} and *fault_LAAPSetpoint* in {omission, valueSubtle, valueCoarse} and *fault_operatorControlLever* in {omission, valueSubtle, valueCoarse} and *fault_LAAPRequest* in {omission, commission});
$\mathbf{G}_{LACU-S1}$:	**always** (*fault_PWMFlow* in {commission} *fault_lockingSwitchPosition* in {commission} and not *fault_PWMFlow_valueCoarse* and not *fault_PWMFlow_valueSubtle* and not *fault_PWMFlow_omission* and not *fault_lockingSwitchPosition_omission*);
$\mathbf{C}_{LACU-S1}$:	For LACU not to exhibit value and omission failure modes, only the specified failure modes can occur on the corresponding input ports.
$\mathbf{E}_{LACU-S1}$:	*name*: LACU FPTC analysis report *supporting argument*: FPTC_rules_conf;
$\mathbf{A}_{LACU-S2}$:	-;
$\mathbf{G}_{LACU-S2}$:	**always** ((*fault_lockingSwitch_noFailure* and *fault_driveDirection_noFailure* and *fault_armPositionAngle1_valueCoarse* and *fault_armPositionAngle2_noFailure* and *fault_LAAPSetpoint_noFailure* and *fault_LAAPRequest_noFailure* and *fault_groundSpeed_noFailure* and *fault_operatorControlLever_noFailure*) **implies** (*fault_lockingSwitchPosition_noFailure* and *fault_PWMFlow_noFailure*))
$\mathbf{C}_{LAAP-S2}$:	The system can handle a standalone value failure of the first arm position sensor without failing itself.
$\mathbf{E}_{LAAP-S2}$:	*name*: Unit testing results *supporting argument*: Unit_test_conf;

Table 8.8: A translated LACU weak contract with the associated evidence

Id	Assertions and support statements
$\mathbf{B}_{LACU-W1}$:	**always** (not *fault_lockingSwitch_commission* and not *fault_groundSpeed_valueSubtle* and not *fault_groundSpeed_valueCoarse* and not *fault_operatorControlLever_valueSubtle* and not *fault_operatorControlLever_valueCoarse* and not *fault_LAAPRequest_commission*);
$\mathbf{H}_{LACU-W1}$:	**always** (not *fault_PWMFlow_commission* and not *fault_lockingSwitchPosition_commission*);
$\mathbf{C}_{LACU-W1}$:	For LACU not to issue any of the commands inadvertently, lockingSwitch and LAAPRequest should not be triggered inadvertently, and groundSpeed and operatorControlLever should not exhibit value failures.
$\mathbf{E}_{LACU-W1}$:	*name*: LACU FPTC analysis report *supporting argument*: FPTC_rules_conf;

hibit failures, which is indicated by the FPTC specifications. Similarly as for the LAAP component, a strong contract for LACU is translated from the FPTC analysis to indicate which failures are mitigated by the component and which can still occur (Table 8.7), while a weak contract is translated to indicate which conditions need to be met for the occurring failures (in this case commission) to be mitigated (Table 8.8).

8.3.5 The resulting argument-fragment

Based on the LACU contract specification compliant to SEooCMM we have instantiated the requirement and contract satisfaction argument patterns and used them to build the argument-fragment that argues absence of the value failure mode in LACU. A part of the resulting argument-fragment is shown in Figure 8.9.

The *AbsValPrimary* goal is supported by the contracts that are related to mitigation of value failures by the LACU, while the *AbsValSecondary* goal addresses the contracts that require the environment of LACU to handle those failure modes that may cause LACU to exhibit a value failure. We identify the strong contract *LACU-S2* as contributing to mitigation of the primary causes of the value failures of LACU, hence its satisfaction is argued under the *ArgAbsValPrimary* goal. Conversely, we use the contract *LACU-S1* to support the *ArgAbsValSecondary* goal, since *LACU-S1* relies on the environment to han-

dle certain failure modes in order for LACU not to exhibit value failures. We use the contract satisfaction pattern introduced in Section 4.2 to develop the contract-specific argument-fragments. Since the *LACU-S2* contract does not have assumptions, its *contractAssume* goal is empty. For the other contract *LACU-S1*, we leave the *LACU-S1_ contractAssume* goal undeveloped as it requires support from the environment of LACU. The refining sub-component contracts of *LACU-S1* and *LACU-S1* include the contracts from LACU subcomponents such as Monitor and LAAP. When developing the LAAP part of the arguments, the artefacts reused with LAAP are used to build that part of the argument.

Since the mitigation of value failures of LACU are handled by both the LACU component and its environment, both the *AbsValPrimary* and *AbsValSecondary* goals are developed. Similarly, when generating an argument based on the HSFM pattern to argue the absence of the commission failures, the *AbsValSecondary* goal would contain the argument over satisfaction of the *LACU-W1* contract. This contract indicates that LACU relies on its environment to contain certain causes of the commission failure in order to mitigate it.

8.3.6 Discussion

A characteristics of FPTC analysis that supports reuse and the reason why we have selected FPTC for failure logic analysis is the possibility to specify FPTC rules for a component in isolation. The support for reuse is based on the assumption that the FPTC rules of the reusable component consider the same set of failure modes as the FPTC rules in the target system. Since the amount of FPTC rules grows exponentially with the increase of input ports of a component, skipping some failure modes on the input ports of such components becomes inevitable [13]. For example, this difficulty was hard to notice on a smaller application example of FLAR2SAF, but when moving to the more realistic LACU case, achieving a sufficiently complete set of FPTC rules became challenging. This is one of the common problems when dealing with similar inductive safety analysis techniques [78].

Not specifying FPTC rules for certain failure modes threatens the support for reuse of the FPTC analysis. Although it is reasonable to skip rules for certain failure modes that might not be possible in certain systems, the fact is that if the component is reused in a system where such failure mode is possible, we cannot afford to assume the failure behaviour of the reused component. Instead of assuming interpretations of the omitted FPTC rules [13], it would be useful to identify if the set of FPTC rules of the reusable component is

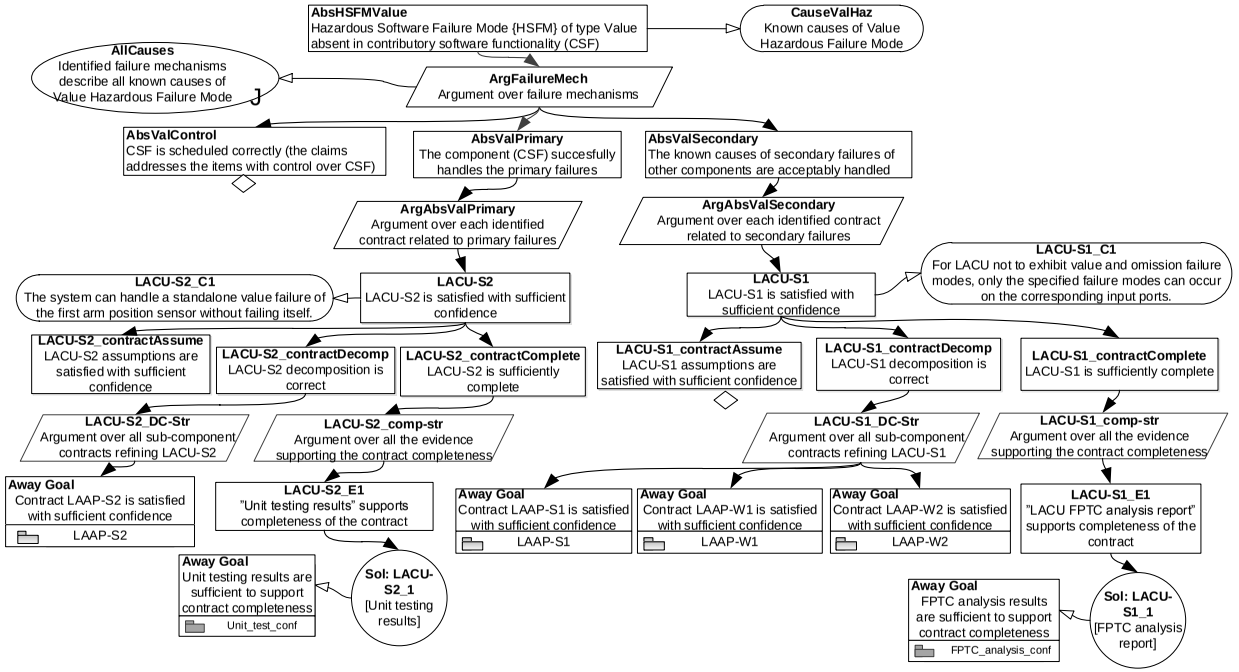

Figure 8.9: Argument-fragment based on the HSFM pattern using the contract satisfaction argument-fragments

sufficient to perform the FPTC analysis in the particular system. This can be done by checking if only the failure modes considered by the FPTC rules of the reused component occur on its input ports in the particular system. If no other failure modes than those considered by the FPTC rules occur on the input ports of the component, then the failure behaviour established by the FPTC rules of that component can be used in the particular system. The strong and weak contracts can be used to achieve this check. As demonstrated in the case study, *capturing the set of considered failure modes in the strong contracts allows us to establish whether the FPTC analysis results achieved in isolation can be reused in the particular system or not*. The strong contract on failure modes alleviates the need for assuming the interpretation of the skipped failure modes by the FPTC rules of a reusable component. One way to handle the situation, where a skipped failure mode occurs on the input of a reused component, would be to design a wrapper or a component similar to the Monitor component of the LACU such that it mitigates or transforms the failure mode not considered. This answers the first research question *RQ2.1* stated in Section 8.1 that FPTC analysis can be performed even though not all failures are considered on inputs of all components, as long as the strong contract assumptions translated with FLAR2SAF are satisfied.

Associating evidence with contracts enables reasoning about reuse of such evidence together with the contracts and utilising such evidence for argument-fragment generation. *SEooCMM enables supporting the contracts, and the failure behaviour they capture, with evidence that provide confidence that the captured failure behaviour is sufficiently correct and complete*. Associating the supporting elements (statements and evidence) to the contracts provides the basis for generating the corresponding argument-fragments. Moreover, since the contracts allow us to distinguish between the primary and secondary failures of the component, *we have demonstrated in the case study that it is possible to generate argument-fragments based on the HSFM argument pattern from such safety contracts*. This answers the second research question *RQ2.2* stated in Section 8.1 that reuse of FPTC-related artefacts is achievable by using FLAR2SAF and SEooCMM.

The generated argument-fragments represent only a portion of the overall argument and can be seen as the skeleton that the overall argument can be built upon. Even after the automated argument-fragment generation, the need for further manual tailoring of the argument remains. The semi-automated nature of such generation of an argument preserves the possibility for customised tailoring of the argument, while enabling benefits of getting a head start by automated generation of parts of the argument. On the scale between full automa-

tion and manual creation of an assurance case argument, FLAR2SAF stands in the middle ground, offering some automation and requiring some manual efforts. The critics of a fully automated argument generation usually point out the issue of validity and veracity of the automatically generated safety arguments from formal models [79], because the arguments are said to be inherently informal. On the other hand, the critics of the manual development of an argument argue that it is a painstaking process of documenting the safety case and it would be better if that effort could be invested in further safety analysis rather than its documentation process [80]. With FLAR2SAF we have opted to take the middle road and automatise portions of the argument and still allow the safety engineer to tailor the informal aspects of the safety argument.

One of the remaining open issues lies in the failure logic analysis itself. The translated contracts and the resulting argument-fragments are as correct and solid as the FLA itself. Establishing the failure behaviour is mainly a manual process that becomes more tedious and error-prone as the size of the component increases, especially if done out-of-context. Relating the expert statements about the failure behaviour of a component directly to the evidence that backs up the expert judgement is a way to increase confidence in the specified failure behaviour. Another issue not covered by the contract translation is the additional assumptions that might have to be made for the evidence used to support the translated contracts. For example, if we have supported a contract with a simulation or a test result, such contract should be enriched to include the assumptions that imply validity of the simulation and the test result.

8.3.7 Validity

Our main focus in this case study was on getting a realistic and sufficiently complex case at a level often found in industry. In cooperation with our industrial partners we have managed this up to a certain point. Although we did not have code behind the system models, we have been able to establish the failure behaviour of the components based on the system description. Since FPTC analysis is useful even before the implementation [13], we have been able to build upon such failure behaviour established without having access to the actual implementation. The downside is that we were unable to fully establish the correctness and completeness of the FPTC rules, which in turn also influenced the completeness and correctness of the contracts.

In our research [81] we applied FLAR2SAF on a simpler system where we assumed that the FPTC rules of the system and the reusable component consider the same set of failure modes. It was apparent that this assumption

is difficult to fulfil when applying FLAR2SAF on a realistic case, as discussed in Section 8.3.6. To weaken this assumption and still make sure that FPTC analysis can be performed, we have introduced an additional type of strong contracts to handle the variable set of the considered failure modes.

In this case study we have been examining feasibility of reuse of FPTC-related safety artefacts. We have not shown the complete set of contracts for the reusable component that is required to check feasibility of reuse of the component itself. For example, to check whether a component is possible to reuse in a particular system there should be a contract to establish whether the value and type of the component ports match with the corresponding ports in its environment. Instead, we have focused on capturing the properties related to reusability of FPTC-related safety artefacts and utilising these artefacts for generation of argument-fragments.

The implications of the results of the case study cannot be generalised to all different reuse scenarios. The feasibility of applying FLAR2SAF to a particular case depends on the case complexity and whether we can establish the failure behaviour of the components in isolation as well as in-context. Still, the case provides evidence for the applicability and usefulness of our approach. Further investigations are needed to allow more general conclusion to be drawn. This includes establishing the level of abstraction at which it is most useful to apply FLAR2SAF. In this particular case we have limited FLAR2SAF application to a portion of a software controller.

Chapter 9

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.

9.1 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 the theory we build upon are presented in Section 2.5. 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. [82] demonstrate how contract-based component specification for different aspects of a component can be expressed using the pattern-based Requirement Specification Language (RSL), where the patterns consist of static text elements and attributes. Furthermore, 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. [83], 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. [84], 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, Söderberg and Johansson [85] propose an approach that uses safety contracts as safety requirements. 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. [86] 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. Westman et al. in another work [87] generalise the established contract theory [10] to environment-centric contracts to provide support for practical engineering and expressing of safety requirements using contracts. The environment-centric contracts relax the constrains on the scope of the assumptions and guarantees beyond the interface of the corresponding component. While environment-centric contracts theory does not distinguish explicitly between the rigid variables such as configuration parameters and other operational variables, Cimatti et al. [66] present a tool-supported contracts-refinement proof system that distinguishes between the two types of variables. Although they can be separately specified, they are treated equally within the contract assumptions, and hence the explicit distinction does not alleviate the challenge contracts have with the different context. In contrast to these works, we emphasise that the distinction to strong and weak contracts is important for addressing the challenge of reuse of a component in different context. Furthermore, in such case, we deem that there should be a 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.

Battram et al. [88] present a method for modular safety assurance based on assumption/guarantee contracts. 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. In contrast to this work, we focus on supporting the out-of-context reasoning and reuse of components together with their contracts and accompanying assurance information across contexts.

Schneider et al. [89] introduce the Digital Dependability Identities (DDIs) as a way to assure dependability of cyber-physical systems. DDIs represent modular, composable and possibly executable specification. Due to many possible configurations of CPS that are not always possible to consider during system design, one of the main goals of DDIs is to provide the basis for run-time certification for the dynamically reconfigurable systems. Conditional Safety Certification (ConSert) represent an initial implementation of DDIs that operate on the level of safety requirements. The conditions in ConSerts are captured between the potentially guaranteed safety requirements (guarantees), and the corresponding demanded safety requirements (demands). While in this thesis we do not focus on runtime assurance, we have explored building upon the assurance aware contract-based design to facilitate runtime assurance similar to ConSerts. Although DDIs main focus is on the run-time certification, the variability offered by the DDIs can be used to achieve reuse. Unlike with DDIs, we aim to generate safety argument-fragments from the system models enriched with contracts, and in that process identify what is relevant in a particular context and include only that information in the resulting argument-fragment.

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 [90] 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 [91]. 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), as proposed by Hawkins [92]. 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. [82] through introduction of strong and weak assumptions, the above mentioned works do not focus on reuse and capturing of safety-relevant information for reusable components. Unlike in our work, none of these works 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.

9.2 Safety Case Artefacts Reuse

There has been many works on modularising representations 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. [93] present an approach to incremental certification that uses “informal” contracts for generation of modular safety case arguments. 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, some of which are tool-supported. The works by Armengaud [94] and Basir et al. [95] focus on automating the compilation of the safety case arguments from pre-existing work products. Denney and Pai [96] 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 automatically generated lower-level argument-fragments derived from formal verification of the implementation against a mathematical specification. The work uses the AutoCert tool for formal verification where the provided specification represent formalised software requirements. Ratiu presents the *Safety.Lab* tool [97] that focuses on model-based safety analysis and generates an argument structure from rich models of various safety-relevant artefacts. Gacek describes the Eclipse-based *Resolute* tool [98] that facilitates generating assurance arguments from architectural models. Nair presents the *Evidence Confidence Assessor (EviCA)* diagramming tool [99] that supports automated generation of confidence arguments related to manually created arguments. Denney and Pai present the *AdvoCATE* [100] toolset includes a variety of automated features for assurance case creation and analysis. AdvoCATE automates instantiation of pre-developed argumentation pattern from a hazard and safety requirement analysis. In contrast to these works, while we also automatically instantiate a pre-developed pattern, we do so from architectural models enriched with assumption/guarantee contracts coupled with safety-relevant artefacts. This allows us to filter the relevant artefacts and provide additional support for reuse and tailoring of context-specific automated argument generation. One of the benefits of implementing our solutions in the AMASS platform is the environment that covers a significant portion of the development cycle. Hence, in such environment we could better connect the system modelling with assurance case modelling, by also allowing traceability between elements in both modelling environments.

A work by Prokhorova [101] 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. [102] propose a model-based approach, model weaving, 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 dependencies 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.

Wardziński and Jones [103] propose an approach similar to model weaving. The proposed approach presents model interfaces that describe system models in a unified way, and a reference table describing argument relations to system models. The goal of the proposed approach is to support interconnection between the assurance case and system modelling by maintaining the consistency of the assurance case and its references to the system model elements with the system models. This approach is more generic than the work presented in this thesis, and proposes a set of high level steps for facilitating integration of system models and assurance case. Conversely, in this thesis we focus on a particular system design technique where we define the relations between assurance case and system models. Thus, we make a concrete proposal for integrating contract-aware system models and assurance case.

The safety standards general lack of detailed guidelines for systematic reuse has triggered researchers to align different reuse engineering methods such as Product-line Engineering (PLE) with the different safety standards. Gallina proposes that PLE can be aligned with the ISO 26262 to facilitate reuse of artefacts [104]. 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. Schulze in another PLE-based approach shows how variability can be integrated into the functional safety models by combining functional safety and variability modelling tools [105]. Hutchesson in his 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 [106]. This work aligns PLE with civil airspace safety standard recommendations on development and integration of reusable elements. Habli [107] 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 as 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 capture variability at the functional level, but also at the assurance level, which helps us promote reuse of safety assets also outside of 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 [108]. 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.

Each of the above approaches offers a way to speed up the creation of the system safety case argument, or its parts, and reuse some of the safety case artefacts in the process. We do not aim at covering the generation of the entire safety argument, nor reuse of all possible artefacts. On the contrary, we focus on assurance when developing systems using contract-based design and how to enhance the reuse capability of contract-based design when using components developed out-of-context. In particular, we do not aim at generating entire arguments, but only fragments that can be used as building blocks for the overall system argument. While some of the works do take in account a kind of contracts for achieving reuse and argument generation, the main focus is generally not on contract-based design and the synergy between system and assurance case modelling. To the best of our knowledge the contribution of our work is in this respect novel and unique.

Chapter 10

Conclusions and future work

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

10.1 Research Goals Revisited

The goal of our research is to facilitate automation of assurance and fine-grained reuse of safety-relevant software components and their accompanying safety case artefacts. As means for achieving our goal we focused on contract-based design that supports independent development of components and compositional verification. We specified three research goals (presented in detail in Section 1.1):

- **Research goal 1**: "To facilitate automated contract-driven assurance in order to reduce the overall assurance efforts."
- **Research goal 2**: "To facilitate reuse of SEooC and their context-specific assurance artefacts by contract-based design."
- **Research goal 3**: "To support reuse of results from existing failure logic analyses and automation of assurance based on those results through the assurance- and reuse aware contract-based design."

To achieve our research goals, we have presented a set of research contributions (briefly summarised and mapped to the research goals in Chapter 1.2):

- **Thesis contribution 1**: "The introduction of argumentation patterns to capture the contract-driven assurance reasoning"
- **Thesis contribution 2**: "Connecting the contract-based system modelling and assurance case modelling on the meta-model level"
- **Thesis contribution 3**: "A method for automated instantiation of the contract-driven assurance argumentation patterns from system models compliant with SEMM"
- **Thesis contribution 4**: "The introduction of strong and weak contracts to manage context variability at the contract level"
- **Thesis contribution 5**: "An extension of SEMM to support context variability across contexts"
- **Thesis contribution 6**: "Alignment of SEooC development with contract-based design assurance and reuse"
- **Thesis contribution 7**: "A method for contract derivation from compositional failure logic analysis"
- **Thesis contribution 8**: "An approach for instantiation of FLA-based argumentation patterns from the derived contracts"

We have performed several industrial case studies on different industrial systems to explore and evaluate the contributions throughout our research. In this thesis, we have detailed two different case studies on the same industrial system where we have grouped the contributions in two integrated solutions, which we have then evaluated in realistic industrial scenarios. In the reminder of the section we describe how we have achieved the thesis research goals through the integration of the different thesis contributions.

Research Goal 1

"To facilitate automated contract-driven assurance in order to reduce the overall assurance efforts."

Component contracts closely relate to system requirements, which are crucial for construction of the system assurance case. When such contracts are specified using a formal or semi-formal notations they can be used to design

the system to meet the specified requirements. We have presented argumentation patterns detailing a strategy to assure the requirements satisfaction using such contracts. To automatically instantiate such argumentation patterns and in that way automatically generate the parts of safety argumentation, a tool supported contract-based design is needed. Furthermore, such a tool needs to support modelling of the specific assurance information together with the contracts and other system modelling elements in order to support automatic instantiation of the proposed requirements and contract assurance argumentation patterns. We propose a Safety Element Meta-Model (SEMM) to connect the system modelling elements, such as components and contracts, with the assurance case modelling elements such as claims and evidence. Having a tool compliant with such enriched meta-model connecting the system and assurance case modelling, we can automatically instantiate the argument patterns for requirements and contract satisfaction assurance.

The problem of automation 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 discussed in [80], the goal of automation is not to replace human reasoning, but to focus it on areas where it is best used. Similarly, in addressing this research goal we are not aiming at eliminating human reasoning from the process of safety reasoning and argumentation. Rather, we aim to support the human reasoning by providing automation of more clerical tasks. In particular, we aim at automating the creation of the skeleton and building blocks of the assurance case, which can be of use to a safety engineer to get a head start in building the overall system assurance case. Furthermore, the goal of assurance aware contract-based design is to enhance information flow from the system engineering and modelling domain to the safety assurance case modelling engineers.

Research Goal 2

"To facilitate reuse of SEooC and their context-specific assurance artefacts by contract-based design."

Contract-based design intrinsically supports reuse and independent development of components. But that support has been mainly focused on components and their implementations, and not that much on the different environments in which those components may be reused, which often imply different safety requirements. The contracts need to be specified such that they provide better support for capturing the variable safety-relevant behaviour across different

environments as well as identifying which of the contract specifications are safety-relevant in a particular environment. To handle the functional variability, these reusable components are often developed with various configurable parameters. These parameters are not only used to tailor the functionality of the components in a particular system, but also to consider safety implications of those functionalities defined by the specific parameters. For example, a particular component configuration may be considered acceptably safe in one environment, but unsafe in another. While contract specifications are primarily focused on resolving the functional variability, by binding the contracts with safety assurance related information, the contracts can assist in identifying the safety assurance information relevant in the specific context.

To support capturing the configurable behaviour exhibited by reusable components, we present strong and weak contracts that allow for capturing component behaviours that are required to hold in all systems in which the component can be used (strong contracts), and 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.

We extend the SEMM meta-model with the strong and weak contracts such that the variability management offered by the strong and weak contracts can be utilised not only for the contract specifications, but also indirectly for the safety assurance information related to those contracts. We refer to the extended meta-model as Safety Element out-of-Context Meta-Model (SEooCMM), as it supports capturing the variability on the system and assurance level, which is essential for development of reusable safety-relevant components such as SEooC. The reuse methodology that we build on top of SEooCMM provides support for fine-grained reuse of components together with their safety assurance information.

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 contract development process to provide such guidelines. We align the proposed process with the automotive ISO 26262 safety standard to show how the safety contract development process can be used to complement an existing safety process. We propose that the safety contract development process is divided into three phases: the preliminary phase where initial strong and weak contracts are captured and matched with safety requirements (done be-

fore the development of the product, hence such contracts may contain speculative/targeted behaviour); the production phase where contracts are actualised with implementation specific-behaviours and supported by evidence (done during/after the development of the product); and the 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.

Research Goal 3

"To support reuse of results from existing failure logic analyses and automation of assurance based on those results through the assurance- and reuse aware contract-based design."

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 [11]. We present a method for deriving safety contracts from Failure Propagation and Transformation Calculus (FPTC) analysis [13] that 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 such a failure could be avoided). Besides the two types of behaviours captured in the contracts, we also capture in the strong contracts the conditions under which the FPTC analysis in isolation has been performed. For example, if we have analysed the component in isolation by only considering coarse and late failure modes on the input ports, when we reuse the component in a system that may propagate other failure modes to

the component, we cannot build upon the previous analysis. In that case, the strong contracts indicate that the analysis on the component level needs to be enriched to consider the new failures modes.

The tool supported assurance and reuse aware contract-based design facilitates generating a set of argument-fragments for each contract and requirement in the system. We show how those argument-fragments can be used to instantiate existing higher-level argumentation patterns to build the overall confidence in the system. In particular, the Handling of Software Failure Modes (HSFM) [71] argument pattern for a component C requires information about known causes of a failure mode and failure mechanisms that address those causes. The failure mechanisms can be classified as: (1) Primary failures within the component C that can cause the failure; (2) secondary failures relating to other components in the system on which the component C depends; and (3) failures caused by the components controlling the component C (e.g., the scheduler).

We use the contracts translated from the FPTC analysis to identify which failure mechanism category the contracts belong to, according to the HSFM pattern. 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. Once we know which contracts belong to which HSFM pattern branch, we use the generated argument-fragments related to those contract to develop the HSFM pattern further.

10.2 Future Research Directions

We have identified several research directions we would like to explore in more depth in the future: *safety contracts language and patterns catalogue*; *multi-concern assurance*; *runtime assurance*; and *Industry 4.0*. In the reminder of this section we briefly summarise each of the future work directions.

10.2.1 Safety contracts language and patterns catalogue

Just as there are templates for specifying requirements, patterns for designing components and argumentation strategies, there are also patterns in specifying contracts to achieve reuse. Using contracts in a specific manner to support reuse naturally leads to identifying those patterns that can speed up the contract specification process. For example, 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 [109]) 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 for reusable components. Furthermore, methods for automatic identification of the assumptions (e.g., by [110]) need to be extended to support identification of the distinct strong and weak assumptions.

10.2.2 Multi-concern assurance

In our current research we have started exploring how assurance and reuse aware contract-based design can provide support in designing systems that need to achieve simultaneously multiple interconnected concerns such as safety and security [111]. The SEooCMM component meta-model that connects system and assurance case modelling domains is limited to a single viewpoint and one type of contracts. To consider multiple viewpoints, we generalise and extend SEooCMM to provide the basis for a generic *Multiple-Viewpoint (*) Safety Element out-of-Context Meta-Model* (*SEooCMM). Since the assurance case is requirements oriented, we define assurance viewpoints in *SEooCMM as sets of requirements. An assurance viewpoint is a generic term that can include different concerns. For example, we can have a safety assurance viewpoint, a security assurance viewpoint etc. Since each requirement can be satisfied by one or more contracts, and each of the contracts can be supported by different evidence, we can automatically build the assurance cases for the different viewpoints through these connections captured in *SEooCMM.

The initial results are promising and show that contracts can be used to capture requirements related to different concerns and facilitate resolving concern variability. In particular, we can generate concern-aware assurance cases by enriching the meta-model that connects the system and assurance case modelling domains so that it includes concern variability. For example, a single assurance case can be generated considering all the different concerns the system is addressing if we define all concerns as part of the same viewpoint. Conversely, if a viewpoint corresponds to a concern, it is also possible to generate concern-specific assurance cases such as a safety case or a security case, which can be useful for the assessors checking compliance to particular concern-specific standards. The work is being performed in the context of the AMASS [74] project and the support for generating concern-specific arguments is being im-

plemented as part of the AMASS platform.

10.2.3 Runtime/Dynamic assurance

While the role of the contracts is clear during design time, its potential use during runtime is yet to be clearly identified. While contracts during runtime can support runtime verification, the assurance aware contracts can also support runtime assurance where we gather operational evidence in our confidence in the contracts. We have started looking into how assurance aware contract-based design can be extended to provide assurance support not only during the pre-operation phase, but also during system runtime. In particular, we currently work on developing a safety assurance concept for Cooperating Cyber-Physical Systems using Wireless Communication [112, 113]. Besides performing standalone functionalities, such systems also join to perform cooperative functionalities such as car platooning. Due to the dynamic nature of the cooperating environment, such cooperative functionalities are not possible to fully assure before deployment. Building upon the assurance and reuse aware contract-based design, we plan to use the contracts to build the confidence for assuring such cooperative functionalities during runtime. To achieve that, we are tagging those contracts that cannot be fully validated during design time, but might need runtime support to assure their confidence continuously. The main component that facilitates such contract-based runtime assurance is the runtime manager component envisaged to perform contract checking during runtime as well as gathering the runtime evidence for such contracts. Having such division on contracts that can be validated fully during design time, and those that cannot, propagates to the assurance case such that we refer to the part of the assurance case that covers the contracts that need runtime assurance as a dynamic/runtime assurance case, while we refer to the part covering the design-time contracts as a static assurance case. The initial results are promising, and we are further iterating this concept and evaluating it in more complex case studies to achieve the desired maturity.

We are also investigating the runtime mechanisms that can facilitate gathering of runtime evidence and means to use such evidence in the assurance case that needs to be continuously assessed in such open systems. In particular, we are looking at the runtime manager component in more detail and what its role could and what it should be in such runtime assurance. We are trying to use such contracts and the runtime manager in assuring smart vehicle functions that need to adapt to the different environments autonomously [114, 115]. In particular, we have started looking into assurance of degradation cascades [114]

using such contracts where we have used the runtime manager as a state manager. As the contracts are describing each degradation mode, checking and resolving the contracts during runtime can be used to indicate if a state change is needed. Also, we are working on generalising the concept of assurance of the degradation cascades to support design and assurance of safety-critical adaptive functions in general [115]. We have used the runtime manager concept as both a state manager and a runtime evidence generator. In a set of wirelessly connected cooperating systems, we use the runtime manager as a smart endpoint node at each vehicle such that the synchronisation of the runtime managers across the set of cooperating systems also manages the state of the overall cooperating system, and not only the local system in which the runtime manager is stored. Further research in this direction is underway, and the evaluation of the concepts is planned on smart vehicle industrial use cases within both the SafeCOP [116] and AMASS [74] projects.

10.2.4 Industry 4.0

One of the characteristics of Industry 4.0 is highly flexible manufacturing where customisation of products for mass production is performed on the fly. Since safety assurance relies on a very strict process of identifying all the conditions that may lead to harm, the flexibility of such manufacturing raises certain challenges in performing safety assurance for such systems. Industry 4.0 is characterised by constant cooperation of various stakeholders/companies/businesses to achieve the desired flexibility. Hence, safety assurance not only needs to be dynamic, but it also needs to be cooperative in the sense that an assurance case cannot be built by a single stakeholder, but they have to constantly work together in assuring the safety of the system. Due to the on the fly customisation, the assurance also needs to be continuous, i.e., at all times we need to be able to assure safety of the system. Furthermore, since we do not have a single delivery of an assurance case, but rather continuous delivery, we need to be able to provide the assurance case on-demand, i.e., an updated assurance case has to be available at any time during the operational life of the system. We have started exploring these challenges [117] and, including identifying how assurance and reuse aware contract-based design can be used as the basis to address these challenges. The research in this direction is underway as part of the Future factories in the Cloud (FiC) [118] project.

Appendices

Appendix A

List of publications (full)

Published material that forms the basis of the thesis

1. *Fostering Reuse within Safety-critical Component-based Systems through Fine-grained Contracts*, Irfan Šljivo, Jan Carlson, Barbara Gallina, Hans Hansson. In: International Workshop on Critical Software Component Reusability and Certification across Domains (CSC2013), Jun 2013.
   Abstract. Our aim is to develop a notion of safety contracts and related reasoning that supports the reuse of software components in and across safety-critical systems, including support for certification related activities such as using the contract reasoning in safety argumentation. In this paper we introduce a formalism for specifying assumption/guarantee contracts for components developed out of context. We are utilising the concepts of weak and strong assumptions and guarantees to customise fine-grained contracts for addressing a broader component context and specification of properties for specific alternative contexts. These out of context contracts can be conveniently instantiated to a specific context, thereby providing support for component reuse.
   Remark: I was the driver of the work under strong supervision of my supervisors listed as co-authors. My contributions include a formalism for specifying assumption/guarantee contracts for components developed out of context. We are utilising the concepts of weak and strong assumptions and guarantees to customise fine-grained contracts for addressing a broader component context and specification of properties for specific alternative contexts. These out of context contracts can be con-

veniently instantiated to a specific context, thereby providing support for component reuse.

2. *Strong and Weak Contract Formalism for Third-Party Component Reuse*, Irfan Šljivo, Barbara Gallina, Jan Carlson, Hans Hansson. In Proceedings of the 3rd International Workshop on Software Certification, 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 a 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.
   Remark: I was the main contributor of the work under the supervision of my supervisors listed as coauthors. The main contribution of the paper is the 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.

3. *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 pa-

per 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.
Remark: 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.

4. *Facilitating Certification Artefacts Reuse Using Safety Contracts*, Irfan Šljivo. In Proceedings of the 14th International Conference on Software Reuse Doctoral Symposium (ICSR DS 2015), January 2015.
Abstract. Safety-critical systems usually need to be certified according a domain-specific safety standard. To reduce the cost and time needed to achieve the safety certification, reuse of certification artefacts together with the corresponding safety-relevant software components is needed. The certification artefacts include safety claims/goals and the supporting evidence that are documented in a safety case. The safety case is required by some safety standards to show that the system is acceptably safe to operate in a given context. Assumption/guarantee contracts can be used to capture the dependencies between a component, including its certification artefacts, and a particular operating context. In this paper we present a research proposal on how safety contracts can be used to facilitate structured reuse of certification-relevant artefacts. More specifically, we explore in which way should such contracts be specified, how can they be derived, and in which way can they be utilised for reuse of safety case argument-fragments and the artefacts those arguments include.
Remark: I was the only contributor of this work. It summarises my licentiate research proposal on how safety contracts can be used to facilitate structured reuse of certification-relevant artefacts. More specifically, we explore in which way should such contracts be specified, how can they be derived, and in which way can they be utilised for reuse of safety case argument-fragments and the artefacts those arguments include.

5. *A Method to Generate Reusable Safety Case Fragments from Compositional Safety Analysis*, Irfan Sljivo, Barbara Gallina, Jan Carlson, Hans Hansson, Stefano Puri. In Proceedings of the 14th International Conference on Software Reuse (ICSR2015), 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.
   Remark: I was the main contributor of the work under supervision of my supervisors also listed as authors of the paper. 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 the CHESS-toolset.

6. *Deriving Safety Contracts to Support Architecture Design of Safety Critical Systems*, Irfan Šljivo, Omar Jaradat, Iain Bate, Patrick Graydon. In Proceedings of the16th IEEE International Symposium on High Assurance Systems Engineering (HASE 2015), 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.
Remark: 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.

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

Remark: 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.

8. *Facilitating Reuse of Safety Case Artefacts Using Safety Contracts*, Irfan Šljivo. Licentiate Thesis. Mälardalen University Press. June 2015.
   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.
   Remark: I was the main contributor of the work. The thesis combined the research done up to that point into a single publication.

9. *Using Safety Contracts to Guide the Integration of Reusable Safety Elements within ISO 26262*, Irfan Šljivo, Barbara Gallina, Jan Carlson,

Hans Hansson. In Proceedings of the 21st IEEE Pacific Rim International Symposium on Dependable Computing (PRDC), IEEE, November 2015.
Abstract. Safety-critical systems usually need to comply with a domain-specific safety standard. To reduce the cost and time needed to achieve the standard compliance, reuse of safety-relevant components is not sufficient without the reuse of the accompanying artefacts. Developing reusable safety components out-of-context of a particular system is challenging, as safety is a system property, hence support is needed to capture and validate the context assumptions before integration of the reusable component and its artefacts in-context of the particular system.We have previously developed a concept of strong and weak safety contracts to facilitate systematic reuse of safety-relevant components and their accompanying artefacts. In this work we define a safety contracts development process and provide guidelines to bridge the gap between reuse of safety elements developed out-of-context of a particular system and their integration in the ISO 26262 safety standard. We use a real-world case for demonstration of the process.
Remark: 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.

10. *Configuration-aware Contracts*. Irfan Šljivo, Barbara Gallina, Jan Carlson, Hans Hansson. In Proceedings of the 4th International Workshop on Assurance Cases for Software-intensive Systems (ASSURE2016), September 2016.
Abstract. Assumption/guarantee contracts represent the basis for independent development of reusable components and their safety assurance within contract-based design. In the context of safety-critical systems, their use for reuse of safety assurance efforts has encountered some challenges: the need for evidence supporting the confidence in the contracts; and the challenge of context, where contracts need to impose different requirements on different systems.In this paper we propose the notion of configuration-aware contracts to address the challenge contract-based design faces with multiple contexts. Since reusable components are often developed with a set of configuration parameters that need to be configured in each context, we extend the notion of contract to distinguish between the configuration parameters and the other variables. Moreover, we define a multi-context reusable component based on the

configuration-aware contracts. Finally, we demonstrate the usefulness of the multi-context components on a motivating case.
Remark: I was the main contributor of the work under supervision of the coauthors. In this paper we propose the notion of configuration-aware contracts to address the challenge contract-based design faces with multiple contexts. Since reusable components are often developed with a set of configuration parameters that need to be configured in each context, we extend the notion of contract to distinguish between the configuration parameters and the other variables. Moreover, we define a multi-context reusable component based on the configuration-aware contracts. Finally, we demonstrate the usefulness of the multi-context components on a motivating case.

11. *A Method to Generate Reusable Safety Case Fragments from Compositional Safety Analysis*, Irfan Šljivo, Barbara Gallina, Jan Carlson, Hans Hansson, Stefano Puri. Journal of Systems and Software: Special Issue on Software Reuse 131, C (September 2017), 570-590.
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 systematic reuse of safety artefacts.To facilitate systematic reuse of safety artefacts we provide a method to generate reusable safety case argument-fragments that include supporting evidence related to compositional safety analysis. The generation is performed from safety contracts that capture safety-relevant behaviour of components in assumption/guarantee pairs backed up by the supporting evidence. We evaluate the feasibility of our approach in a real-world case study where a safety related component developed in isolation is reused within a wheel-loader.
Remark: I was the main contributor of the work under supervision of my supervisors included in the co-authors list. My contributions include extension of the Safety Element out-of-Context Meta-model, 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 validation of the feasibility of the proposed approach in a real-world case

study. Stefano Puri provided valuable comments about the proposed approach. Moreover, he provided support for using the CHESS-toolset for running FPTC analysis and modelling the system that was used in the case study. Furthermore, he was involved in implementing some of the results in the CHESS-toolset.

12. *Tool-Supported Safety-Relevant Component Reuse: From Specification to Argumentation*, Irfan Šljivo, Barbara Gallina, Jan Carlson, Hans Hansson, Stefano Puri. In Proceedings of the 23rd International Conference on Reliable Software Technologies (Ada-Europe), June 2018.
Abstract. Contracts are envisaged to support compositional verification of a system as well as reuse and independent development of their implementations. But reuse of safety-relevant components in safety-critical systems needs to cover more than just the implementations. As many safety-relevant artefacts related to the component as possible should be reused together with the implementation to assist the integrator in assuring that the system they are developing is acceptably safe. Furthermore, the reused assurance information related to the contracts should be structured clearly to communicate the confidence in the component. In this work we present a tool-supported methodology for contract-driven assurance and reuse. We define the variability on the contract level in the scope of a trace-based approach to contract-based design. With awareness of the hierarchical nature of systems subject to compositional verification, we propose assurance patterns for arguing confidence in satisfaction of requirements and contracts. We present an implementation extending the AMASS platform to support automated instantiation of the proposed patterns, and evaluate its adequacy for assurance and reuse in a real-world case study.
Remark: I was the main contributor of the work under supervision of my supervisors included in the co-authors list. My contributions include the development of the theoretical concepts as well as the implementation of the automated transformations between OCRA and CHESS on the one hand, and CHESS and OpenCert on the other hand. Stefano Puri contributed with CHESS-specific implementations as well as discussions on the theoretical concepts related to system modelling and combination with assurance case modelling.

Published material that forms the initial research of the future directions

1. *Building Multiple-Viewpoint Assurance Cases Using Assumption/Guarantee Contracts.* Irfan Sljivo, Barbara Gallina. In: 1st International workshop on Interplay of Security, Safety and System/Software Architecture (ISSA-2016), November 2016.
   Abstract. Assurance cases in form of structured arguments are often required by standards to show that a system is acceptable for its intended purpose with respect to a particular assurance viewpoint such as safety or security. The goal of such a case is to present an argument that connects the requirements of a particular viewpoint with the supporting evidence. Building a set of assurance cases for the different viewpoints can be time-consuming and costly. Means are needed to automate and reuse the assurance case artefacts between the assurance cases for the different viewpoints.In this paper we present how assumption/guarantee contracts can be used to facilitate reuse of assurance case artefacts by building multiple-viewpoint assurance cases from the contracts. More specifically, we build upon the previous work on argument-fragment generation from such contracts to allow for generating viewpoint specific argument-fragments. We illustrate the approach on a motivating case.
   Remark: In this paper we present how assumption/guarantee contracts can be used to facilitate reuse of assurance case artefacts by building multiple-viewpoint assurance cases from the contracts. More specifically, we build upon the previous work on argument-fragment generation from such contracts to allow for generating viewpoint specific argument-fragments. We illustrate the approach on a motivating case.

2. *Cooperative Safety Critical CPS Platooning in SafeCOP.* Samer Medawar, Detlef Scholle, Irfan Sljivo. In: 5th EUROMICRO/IEEE Workshop on Embedded and Cyber-Physical Systems (ECYPS-2017), June 2017.
   Abstract. This paper presents the platooning research within the Safe Cooperating Cyber-Physical Systems using Wireless Communication (SafeCOP) project. Cooperating Cyber-Physical Systems (CO-CPS) using wireless communication and having multiple stakeholders, dynamic system definitions (openness), and unpredictable operating environments, are the main application of SafeCOP. In addition to safety assurance methods and tools, SafeCOP devises a runtime manager architecture that detects irregular operation, hence, prompting a safe degraded mode in

case of need.SafeCOP lays a safety and security umbrella over the usage of current wireless technologies, contributes to new standards and regulations by providing scientifically validated solutions to establish standards which also addresses cooperation and system-of-systems issues. SafeCOP addresses several use cases that solve customer related problems. However, in this paper we will present a use case that extract generic principles from the combination of the previous use cases to stimulate the European collaboration around the project objectives, and to collect general requirements for the SafeCOP solution, applicable across all the areas considered. We consider a CO-CPS composed of two or more systems moving in a platoon while cooperating in a safe function.
Remark: In this paper we elaborate the role of the runtime manager and its relation to contracts in the SafeCOP architecture. We demonstrate on a platooning use case how such runtime manager can be used for continuous assurance of cooperative cyber-physical systems.

3. *Safe Cooperating Cyber-Physical Systems using Wireless Communication.* Paul Pop, Detlef Scholle , Irfan Sljivo, Hans Hansson, Gunnar Widforss, Malin Rosqvist. In: Elsevier journal of Microprocessors and Microsystems (MICPRO), July 2017.
Abstract. This paper presents an overview of the ECSEL project entitled "Safe Cooperating Cyber-Physical Systems using Wireless Communication" (SafeCOP), which runs during the period 2016?2019. SafeCOP targets safety-related Cooperating Cyber-Physical Systems (CO-CPS) characterised by use of wireless communication, multiple stakeholders, dynamic system definitions (openness), and unpredictable operating environments. SafeCOP will provide an approach to the safety assurance of CO-CPS, enabling thus their certification and development. The project will define a runtime manager architecture for runtime detection of abnormal behaviour, triggering if needed a safe degraded mode. SafeCOP will also develop methods and tools, which will be used to produce safety assurance evidence needed to certify cooperative functions. SafeCOP will extend current wireless technologies to ensure safe and secure cooperation, and also contribute to new standards and regulations, by providing certification authorities and standardization committees with the scientifically validated solutions needed to craft effective standards extended to also address cooperation and system-of-systems issues. The project has 28 partners from 6 European countries, and a

budget of about 11 million Euros corresponding to about 1,300 person-months.
Remark: In this paper we have presented the SafeCop Cooperative Safety Assurance Concept that includes the usage of the assumption and guarantee contracts for achieving dynamic/runtime assurance. We propose the concept of a runtime manager that has the aim to evaluate the contracts during runtime and gather runtime evidence that should be used to evaluate confidence in the contracts during the system operation.

4. *Assuring Degradation Cascades of Car Platoons via Contracts*, Irfan Šljivo, Barbara Gallina, Bernhard Kaiser. In Proceedings of the 6th International Workshop on Next Generation of System Assurance Approaches for Safety-Critical Systems (SASSUR-2017), September 2017.
Abstract. Automated cooperation is arriving in practice, for instance in vehicular automation like platoon driving. The development and safety assurance of those systems poses new challenges, as the participating nodes are not known at design time; they engage in communication at runtime and the system behaviour can be distorted at any time by failures in some participant or in the communication itself. When running on a highway, simply switching off the function is not an option, as this would also result in hazardous situations. Graceful degradation offer a systematic approach to define a partial-order of less and less acceptable operation modes, of which the best achievable is selected in presence of failures.In this work we propose an approach for assurance of the degradation cascades based on mode-specific assertions, captured by assumption/guarantee contracts. More specifically, we share our experiences and methodology for specifying the contracts for both the nominal safe behaviour as well as the less safe but acceptable behaviour in presence of failures. Furthermore, we present an argument pattern for adequacy of the degradation cascades for meeting the global safety goals based on the contracts. We illustrate our approach by a car platooning case study.
Remark: In this paper we have proposed how to use the assumption and guarantee contracts and the assurance aware contract-based design to assure degradation cascades of car platoons.

5. *Challenges of Safety Assurance for Industry 4.0.* Omar Jaradat, Irfan Sljivo, Ibrahim Habli , Richard Hawkins. In Proceedings of the 13th European Dependable Computing Conference (EDCC'17), September 2017.
Abstract. The Internet-of-Things (IoT) has enabled Industry 4.0 as a

new manufacturing paradigm. The envisioned future of Industry 4.0 and Smart Factories is to be highly configurable and composed mainly of the 'things' that are expected to come with some, often partial, assurance guarantees. However, many factories are categorised as safety-critical, e.g. due to the use of heavy machinery or hazardous substances. As such, some of the guarantees provided by the 'things', e.g. related to performance and availability, are deemed as necessary in order to ensure the safety of the manufacturing processes and the resulting products. In this paper, we explore key safety challenges posed by Industry 4.0 and identify the characteristics that its safety assurance should exhibit. We propose a set of safety assurance responsibilities, e.g. system integrators, cloud service providers and 'things' suppliers. Finally, we reflect on the desirable modularity of such a safety assurance approach as a basis for cooperative, on-demand and continuous reasoning for Industry 4.0 architectures and services.
Remark: In this paper we explore the challenges of assurance for Industry 4.0 where the open and dynamically reconfigurable systems require continuous and on-demand assurance. We explore how contract-based design can assist in addressing some of those challenges.

6. *Contract-Based Assurance for Wireless Cooperative Functions of Vehicular Systems.* Svetlana Girs, Irfan Sljivo, Omar Jaradat. In Proceedings of the 43rd Annual Conference of the IEEE Industrial Electronics Society (IECON 2017). October 2017.
Abstract. Cooperation of vehicular systems is the stepping stone towards both road and indoor smart transportation systems. It aims at increasing transportation efficiency and safety compared to the stand-alone vehicular systems. The usage of wireless communication as the foundation of such safety-critical cooperation needs to be embraced with all its benefits and flaws compared to the wired communication. The cooperative functions need to be designed to adapt to the varying reliability of the wireless communication channels such that both the stand-alone vehicles as well as the smart transportation system formed by their cooperation are deemed sufficiently safe. In this paper we build upon a contract-based runtime monitoring architecture and propose a methodology for assuring adaptive behaviour of transportation with respect to the wireless communication channel failures. More specifically, we elaborate how safety analysis of the interaction of the wirelessly connected vehicles can be used as the basis for derivation of the adaptive modes and the

corresponding contracts. Furthermore, we discuss how such contracts can be used as the basis for assurance of the adaptive wireless cooperation. We illustrate the proposed methodology on a smart transportation system of a factory.
Remark: In this paper we try to generalise the work on assurance of degradation cascades using contracts to support assurance of different mode-switching applications, especially of the safety-critical wireless cooperative functions of vehicular systems.

Additional publications

1. *Towards a Safety-oriented Process Line for Enabling Reuse in Safety Critical Systems Development and Certification.* Barbara Gallina, Irfan Sljivo, Omar Jaradat. In Proceedings of the 35th Annual Software Engineering Workshop (ISOLA workshop) (SEW 2012), IEEE, October 2012. **Remark:** In this paper, we adopt process line approach in the framework of safety processes. This means that we treat a family of processes as a product line, and we identify commonalities and variabilities between them. The resulting information guides developers in reusing parts of the process, the system and safety case, e.g. which parts to make more generic, isolating changes in others to avoid ripple effects etc.

2. *Towards Cloud-Based Enactment of Safety-Related Processes.* Sami Alajrami, Barbara Gallina, Irfan Sljivo, Alexander Romanovsky, Petter Isberg. In Proceedings of the 35th International Conference on Computer Safety, Reliability and Security (SafeComp2016), September 2016. **Remark:** This work adapts previous work on cloud-based software engineering by enriching the architecture with an automatic support for generation of both, product-based safety arguments from failure logic analysis results and process-based arguments from the process model and the enactment data. The approach is demonstrated using a fragment of a process adapted from the aerospace domain.

3. *Agent-centred Approach for Assuring Ethics in Dependable Service Systems.* Irfan Sljivo, Elena Lisova, Sara Afshar. In Proceedings of the 13th IEEE World Congress on Services (SERVICES-2017), June 2017. **Remark:** In this work we propose an agent-centred approach for assuring ethics in dependable technological service systems. We build upon assurance of safety and security and propose the notion of ethics assurance

case as a way to assure that individual users have been made aware of all the ethically challenging decisions that might be performed or enabled by the service provider. We propose a framework for identifying and categorising ethically challenging decisions, and documenting the ethics assurance case. We apply the framework on an illustrative example.

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] NATO RTO Task Group IST-027/RTG-009. Validation, Verification and Certification of Embedded Systems. ISRN RTO-TR-IST-027. 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] ISO 26262-10. *Road vehicles — Functional safety — Part 10: Guideline on ISO 26262*. International Organization for Standardization, 2011.

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

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

[8] 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.

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

[10] Albert Benveniste, Benoit Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Thomas Henzinger, and Kim G. Larsen. Contracts for System Design. Research Report RR-8147, Inria, November 2012.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

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

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

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

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

[22] 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.

[23] 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.

[24] Object Management Group (OMG). SACM: Structured Assurance Case Metamodel. Technical report, Version 2.0, OMG, 2018. http://www.omg.org/spec/SACM.

[25] C Michael Holloway. Safety case notations: Alternatives for the non-graphically inclined? In *System Safety, 2008 3rd IET International Conference on*, pages 1–6. IET, 2008.

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

[27] GSN Community Standard Version 2. Technical report, Assurance Case Working Group of The Safety-Critical Systems Club, January 2018.

[28] 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.

[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). *ED79/ARP-4754: Certification Considerations for Highly-integrated or Complex Aircraft Systems*. Society of Automotive Engineers, 1996.

[36] 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.

[37] 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.

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

[39] ISO 26262-3:2011. *Road vehicles — Functional safety — Part 3: Concept phase*. International Organization for Standardization, 2011.

[40] ISO 26262-4:2011. *Road vehicles — Functional safety — Part 4: Product development at the system level*. International Organization for Standardization, 2011.

[41] ISO 26262-5:2011. *Road vehicles — Functional safety — Part 5: Product development at the hardware level*. International Organization for Standardization, 2011.

[42] ISO 26262-6:2011. *Road vehicles — Functional safety — Part 6: Product development at the software level*. International Organization for Standardization, 2011.

[43] ISO 26262-7:2011. *Road vehicles — Functional safety — Part 7: Production and operation*. International Organization for Standardization, 2011.

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

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

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

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

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

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

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

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

[60] 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.

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

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

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

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

[65] 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.

[66] A. Cimatti and S. Tonetta. Contracts-refinement proof system for component-based embedded systems. *Science of Computer Programming*, 97(3):333–348, 2014.

[67] H. Martin et al. "Demonstration report of the automotive and construction equipment use cases , Version 1.0" SafeCer, Deliverable D503.1, November 2015.

[68] Bishop, P. and Bloomfield, R. A Methodology for Safety Case Development. In Redmill, F. and Anderson, T., editor, *Industrial Perspectives of Safety-critical Systems: 6th Safety-critical Systems Symposium*, pages 194–203. Springer, 1998.

[69] Marc Bender, Tom Maibaum, Mark Lawford, and Alan Wassyng. Positioning Verification in the Context of Software/System Certification. *Electronic Communications of the EASST*, 46, 2011.

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

[71] R. Weaver, J. McDermid, and T. Kelly. Absence of Value Hazardous Failure Mode, http://www.goalstructuringnotation.info/archives/220, 2004.

[72] B. Gallina, M. A. Javed, F. U. Muram, and S. Punnekkat. Model-driven Dependability Analysis Method for Component-based Architectures. In *38th Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, 2012.

[73] B. Gallina and S. Punnekkat. FI^{4}FA: A Formalism for Incompletion, Inconsistency, Interference and Impermanence Failures Analysis. In *2nd International Workshop on Distributed Architecture modeling for Novel Component based Embedded systems*. IEEE, 2011.

[74] ECSEL-JU-692474. AMASS – Architecture-driven, Multi-concern and Seamless Assurance and Certification of Cyber-Physical Systems. http://www.amass-ecsel.eu/.

[75] Object Management Group (OMG). MOFM2T: MOF Model to Text Transformation Language. Technical report, V1.0, OMG. http://www.omg.org/spec/MOFM2T/1.0/, 2008.

[76] R. Stake. Case Studies. In N. K. Denzin and Y. S. Lincoln, editors, *Handbook of Qualitative Research*, chapter 14, pages 236–247. Sage Publications, Oxford, 1994.

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

[78] David Parker, Martin Walker, and Yiannis Papadopoulos. Model-based functional safety analysis and architecture optimisation. *Embedded Computing Systems: Applications, Optimization, and Advanced Design: Applications, Optimization, and Advanced Design*, pages 79–92, 2013.

[79] Ibrahim Habli and Tim Kelly. Balancing the Formal and Informal in Safety Sase Arguments. In *VeriSure: Verification and Assurance Workshop, colocated with Computer-Aided Verification (CAV)*, July 2014.

[80] J. Rushby. Logic and Epistemology in Safety Cases. In *32nd International Conference on Computer Safety, Reliability, and Security*, volume 8153 of *Lecture Notes in Computer Science*, pages 1–7. Springer, September 2013.

[81] 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, January 2015.

[82] 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.

[83] 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.

[84] 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.

[85] 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.

[86] 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.

[87] J. Westman and M. Nyberg. Environment-Centric Contracts for Design of Cyber-Physical Systems. In *Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014*, volume 8767 of *Lecture Notes in Computer Science*, pages 218–234. Springer, 2014.

[88] 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.

[89] D. Schneider, M. Trapp, Y. Papadopoulos, E. Armengaud, M. Zeller, and K. Höfig. WAP: Digital dependability identities. In *26th International Symposium on Software Reliability Engineering*, pages 324–329. IEEE, 2015.

[90] 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.

[91] 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.

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

[93] 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.

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

[95] 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.

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

[97] Daniel Ratiu, Marc Zeller, and Lennart Killian. Safety.lab: Model-based domain specific tooling for safety argumentation. In *International Conference on Computer Safety, Reliability, and Security*, volume 9338 of *LNCS*, pages 72–82. Springer, 2015.

[98] Andrew Gacek, John Backes, Darren Cofer, Konrad Slind, and Mike Whalen. Resolute: an assurance case language for architecture models. *ACM SIGADA Ada Letters*, 34(3):19–28, December 2014.

[99] Sunil Nair, Neil Walkinshaw, Tim Kelly, and Jose Luis de la Vara. An evidential reasoning approach for assessing confidence in safety evidence. In *26th International Symposium on Software Reliability Engineering*, pages 541–552. IEEE, 2015.

[100] Ewen Denney and Ganesh Pai. Tool support for assurance case development. *Automated Software Engineering*, Dec 2017.

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

[102] 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.

[103] Andrzej Wardziński and Paul Jones. Uniform model interface for assurance case integration with system models. In *International Conference on Computer Safety, Reliability, and Security*, pages 39–51. Springer, 2017.

[104] B. Gallina, A. Gallucci, K. Lundqvist, and M. Nyberg. VROOM & cC: a Method to Build Safety Cases for ISO 26262-compliant Product Lines. In *2nd Workshop on Next Generation of System Assurance Approaches for Safety-Critical Systems*. Hyper Articles en Ligne (HAL), September 2013.

[105] 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.

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

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

[108] 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.

[109] 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.

[110] 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.

[111] Irfan Sljivo and Barbara Gallina. Building multiple-viewpoint assurance cases using assumption/guarantee contracts. In *1st International workshop on Interplay of Security, Safety and System/Software Architecture*, November 2016.

[112] Paul Pop, Detlef Scholle, Irfan Sljivo, Hans Hansson, Gunnar Widforss, and Malin Rosqvist. Safe cooperating cyber-physical systems using wireless communication. *Elsevier journal of Microprocessors and Microsystems*, 53:42–50, July 2017.

[113] Samer Medawar, Irfan Sljivo, and Detlef Scholle. Cooperative safety critical cps platooning in safecop. In *5th EUROMICRO/IEEE Workshop on Embedded and Cyber-Physical Systems*, June 2017.

[114] Irfan Sljivo, Barbara Gallina, and Bernhard Kaiser. Assuring degradation cascades of car platoons via contracts. In Friedemann Bitsch Stefano Tonetta, Erwin Schoitsch, editor, *6th International Workshop on Next Generation of System Assurance Approaches for Safety-Critical Systems*, volume 10489, pages 317–329. Springer, September 2017.

[115] Svetlana Girs, Irfan Sljivo, and Omar Jaradat. Contract-based assurance for wireless cooperative functions of vehicular systems. In *43rd Annual Conference of the IEEE Industrial Electronics Society*, October 2017.

[116] ECSEL-JU-692529. SafeCOP – Safe Cooperating Cyber-Physical Systems using Wireless Communication. http://www.safecop.eu/.

[117] Omar Jaradat, Irfan Sljivo, Ibrahim Habli, and Richard Hawkins. Challenges of safety assurance for industry 4.0. In *European Dependable Computing Conference*. IEEE Computer Society, September 2017.

[118] Swedish Fondation for Strategic Research (SSF). FiC – Future factories in the Cloud. http://www.es.mdh.se/fic/.

MÄLARDALEN UNIVERSITY
SWEDEN

Address: P.O. Box 883, SE-721 23 Västerås. Sweden
Address: P.O. Box 325, SE-631 05 Eskilstuna. Sweden
E-mail: info@mdh.se **Web:** www.mdh.se

ISBN 978-91-7485-401-5
ISSN 1651-4238