A Method to Generate Reusable Safety Case Argument-Fragments from Compositional Safety Analysis

Irfan Šljivo[a,*], Barbara Gallina[a], Jan Carlson[a], Hans Hansson[a], Stefano Puri[b]

[a]*Mälardalen Real-Time Research Centre, Mälardalen University, Högskoleplan 1, Västerås, Sweden*
[b]*Intecs, SpA, Pisa, Italy*

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.

Keywords: Component-based architectures, Contract-based architectures, Compositional safety analysis, Modular argumentation, Safety argumentation reuse

*Corresponding author

Email addresses: irfan.sljivo@mdh.se (Irfan Šljivo), barbara.gallina@mdh.se (Barbara Gallina), jan.carlson@mdh.se (Jan Carlson), hans.hansson@mdh.se (Hans Hansson), stefano.puri@intecs.it (Stefano Puri)

1. Introduction

A recent study within the US Aerospace Industry shows that reuse is more present when developing embedded systems than non-embedded systems [1]. The study reports that code is reused most of the time, followed by requirements and architectures in significantly smaller scale than code. Aerospace industry, as most other safety-critical industries such as automotive, needs to follow a domain specific safety standard that requires additional artefacts to be provided alongside the code to show that the code is acceptably safe to operate in a given context. The costs of producing the verification artefacts are estimated at more than 100 USD per code line, while for highly critical applications the costs can reach up to 1000 USD per line [2]. We refer to the process of achieving compliance with a particular standard as the certification process.

In most cases, the certification efforts are required to include a safety case, which is rather time-consuming and expensive task to provide. A safety case is documented in form of an explained and well-founded structured argument to clearly communicate that the system is acceptably safe to operate in a given context [3]. While a safety case includes all the artefacts (e.g., code, requirements, results of failure analyses or verification evidence) produced during the system lifecycle, a safety case argument represents means to communicate the reasoning behind the safety case to why the system is acceptably safe to operate in a given context (Figure 1).

Most safety standards are starting to acknowledge the need for reuse, hence the latest versions of both aerospace (DO-178C) and automotive (ISO 26262) industry standards explicitly support notions that enable reuse, e.g., the notion of Safety Element out of Context (SEooC) within automotive [4] and Reusable Software Components (RSC) within aerospace [5]. This allows for easier integration of reusable components, such as Commercial off the shelf (COTS), but it also means that some safety artefacts of the reused components should be reused as well if we are to fully benefit from the reuse and safely integrate the reused component into the new system.

The difficulty that hinders reuse within safety-critical systems is that safety is a system property. This means that hazard analysis and risk assessment used to analyse what can go wrong at system level, as required by the standards, can only be performed in a context of the specific system.

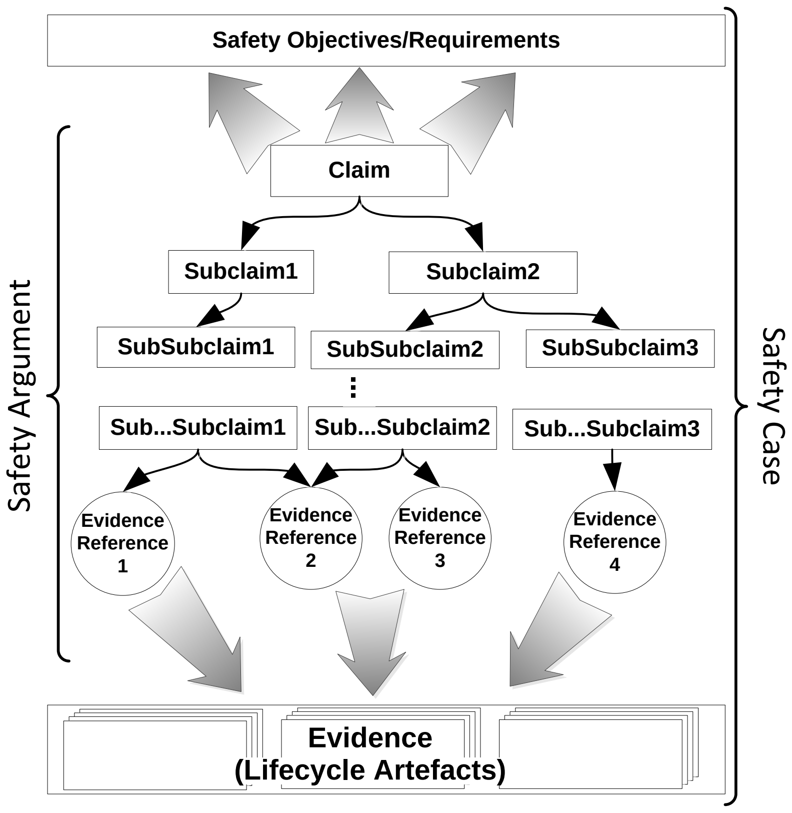


Figure 1: The role of safety argumentation within a safety case

To overcome this difficulty compositional approaches are needed. CHESS-FLA [6] is a plugin within the CHESS toolset[1] that enables execution of Failure Logic Analysis (FLA) such as Fault Propagation and Transformation Calculus (FPTC). FPTC allows us to calculate system level behaviour given the behaviour of the individual components established in isolation in form of FPTC rules. Such compositional failure analyses enable reuse of safety artefacts within safety-critical systems.

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

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

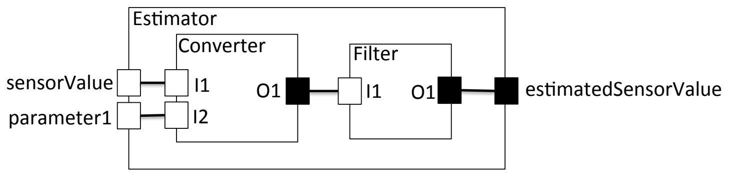


Figure 2: Running example

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

Systematic reuse of safety case artefacts can be achieved by generating artefacts for a specific system from specifications written in a domain specification language, often referred to as generative reuse [7]. For example, a safety-relevant component developed out-of-context together with a safety argument is reused in a particular system. Since such an argument could contain information that might be irrelevant for the particular system in which the component is reused, system-specific information should be captured in specifications so that system-specific safety argument could be generated for the particular system.

In our previous work we developed the notion of safety contracts related to software components to promote reuse of the components developed out-of-context together with their certification data [8]. Moreover, we have proposed a (semi)automatic method to generate argument-fragments for the software components from their associated safety contracts [9]. In this work we propose a method called FLAR2SAF that uses failure logic analysis results (FLAR) to generate safety case argument-fragments (SAF). More specifically, we derive safety contracts for a component from FLAR. Then, we adapt our method for generation of argument-fragments to provide better support for reuse of the argument-fragments and the evidence they contain.

In particular, the input/output behaviour of a component developed out-of-context can be captured by FPTC rules. Figure 2 shows a running example we will use throughout the paper. 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 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: $I1.valueCoarse \rightarrow O1.noFailure$. We can use these behaviours obtained by FPTC analysis to derive safety contracts that can be further supported by evidence and used to form clear and comprehensive argument-fragments. For example, if 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. For generating argument-fragments associated to the failure behaviour of the components we use an established argument pattern [10].

The main contribution of this paper is a method for the design and preparation for certification of reusable COTS-based safety-critical architectures. First, we propose a Safety Element out-of-context Meta-Model (SEooCMM) aligned with the standardised argumentation and evidence meta-models. Next, we provide a conceptual mapping of FPTC rules to safety contracts. Finally, we extend the argument-fragment generation method to generate reusable argument-fragments based on an existing argumentation pattern. We evaluate the feasibility of our approach in a real-world case study. .

1.1. Outline

The rest of the paper is organised as follows: In Section 2 we provide background information. We present the SEooCMM in more detail in Section 3. In Section 4 we present the rationale behind our approach and methods to derive safety contracts from FPTC analysis and generate corresponding argument-fragments. In Section 5 we evaluate the feasibility of FLAR2SAF by applying it to a real-world case. We present the related work in Section 6, and conclusions and future work in Section 7.

2. Background

In this section we briefly provide some background information on COTS-based safety-critical architectures and safety contracts. Furthermore, we recall essential information concerning the CHESS-FLA plugin within the CHESS toolset, together with a brief introduction to safety cases, safety case modelling and the ISO 26262 safety standard. Finally, we present the standardised assurance case argumentation and evidence meta-models.

2.1. COTS-based safety-critical architectures

In the context of safety critical systems, COTS-driven development is becoming more and more appealing. The typical V model that constitutes the reference model for various safety standards is being combined with the typical component-based development. As Figure 3 depicts, the top-down and bottom-up approach meet in the gray zone. Initially a top-down approach is carried out. The typical safety process starts with hazards identification which is conducted by analysing (brainstorming on) failure propagation, based on an initial description of the system and its possible functional architecture. If a failure at system level may lead to intolerable hazards, safety requirements are formulated 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). More specifically, once the safety requirements are decomposed onto components (hardware/software), COTS (developed via a bottom-up approach) can be selected to meet those requirements. If the selected components do not fully meet the requirements, some adaptations can be introduced.

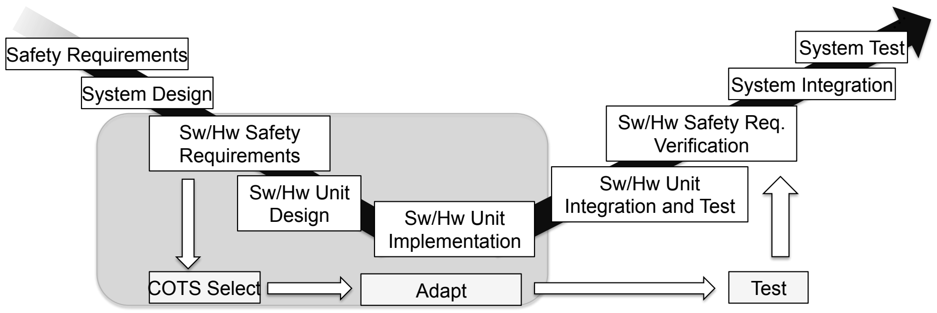

Safety Requirements
System Design
Sw/Hw Safety Requirements
Sw/Hw Unit Design
Sw/Hw Unit Implementation
Sw/Hw Unit Integration and Test
Sw/Hw Safety Req. Verification
System Integration
System Test
COTS Select
Adapt
Test

Figure 3: Safety-critical system development/COTS-driven development

To ease the selection of components, contracts play a crucial role. In our previous work, we have proposed a contract-based formalism with strong $\langle A, G\rangle$ and weak $\langle B, H\rangle$ contracts to distinguish between context-specific properties and those that must hold for all contexts [8]. A traditional component contract $C = \langle A, G\rangle$ is composed of assumptions (A) on the environment of the component and guarantees (G) that are offered by the component if

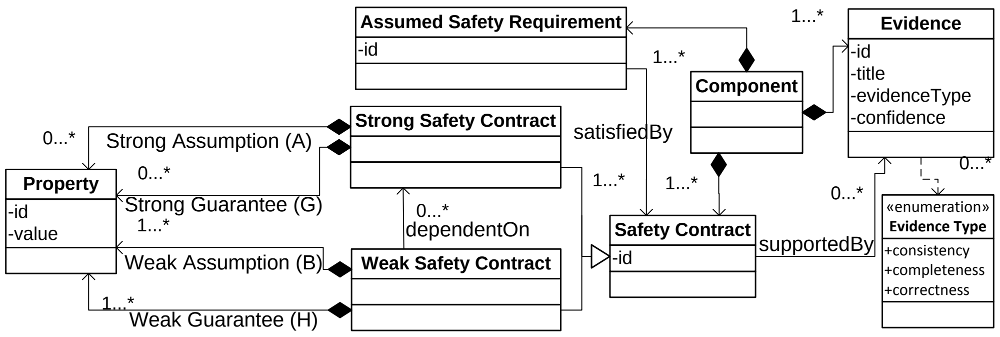


Figure 4: Component and safety contract meta-model [9]

the assumptions are met. The strong contract assumptions (A) are required to be satisfied in all contexts in which the component is used, hence the corresponding strong guarantees (G) are offered in all contexts in which the component can be used. For example, a strong assumption could be the minimum amount of memory a component requires to operate. The weak contract guarantees (H) are offered only in those contexts where, in addition to the strong assumptions, the corresponding weak assumptions (B) are satisfied as well. This makes the weak contracts context specific, e.g., a timing behaviour of a component on a specific platform could be captured by a weak contract.

We denote a contract capturing safety-relevant behaviour as a safety contract. In our previous work [9] we introduced a component meta-model (Figure 4) that connects safety contracts with supporting evidence, which provides a base for evidence artefact reuse together with the contracts. The component meta-model specifies a component in an out-of-context setting composed of safety contracts, evidence and the assumed safety requirements. Each safety requirement is satisfied by at least one safety contract, and each contract can be supported by one or more evidence. For example, if we assume that a value failure on the output of the component can be hazardous, then we define an assumed safety requirement that specifies that value failures should be appropriately handled. This requirement is addressed by a contract that captures in its assumptions the identified properties that need to hold for the component to guarantee that the value failure is appropriately handled. If such a contract is derived from FPTC analysis, then we can further support the contract with the analysis results.

2.2. CHESS-FLA within the CHESS toolset

CHESS-FLA [6] is a plugin within the CHESS toolset that includes two Failure Logic Analysis (FLA) techniques:

- Fault Propagation and Transformation Calculus (FPTC) [11] - a compositional technique to qualitatively assess the dependability of component-based systems, and
- A Formalism for Incompletion, Inconsistency, Interference and Impermanence Failures' Analysis (FI^{4}FA) [12] - an FPTC extension that allows for analysis of mitigation behaviour in the specific context of transaction-based computations.

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

Input failures are assumed to be propagated or transformed deterministically, i.e., for a combination of failures on the input, there can be only one combination of failures on the output.

The syntax supported in CHESS-FLA to specify the FPTC rules is shown in Figure 5. The example of a compliant expression "$I1.valueCoarse \rightarrow O1.noFailure$" mentioned in Section 1 demonstrates the sink behaviour of the Filter component (Figure 2) and should be read as follows: if the component receives on its input port I1 a coarse (i.e. clearly detectable) value

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

Figure 5: FPTC syntax supported in CHESS-FLA

failure (a failure that manifests itself as a failure mode by exceeding the allowed range), it generates no failure on its output port O1.

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 [11]. 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 2) 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.: "$I1.omission, I2.wildcard \rightarrow O1.omission$".

2.3. Safety cases and safety case modelling

A Safety case in form of an explained (argued about) and well-founded (evidence-based) structured argument is often required to show that the sys-

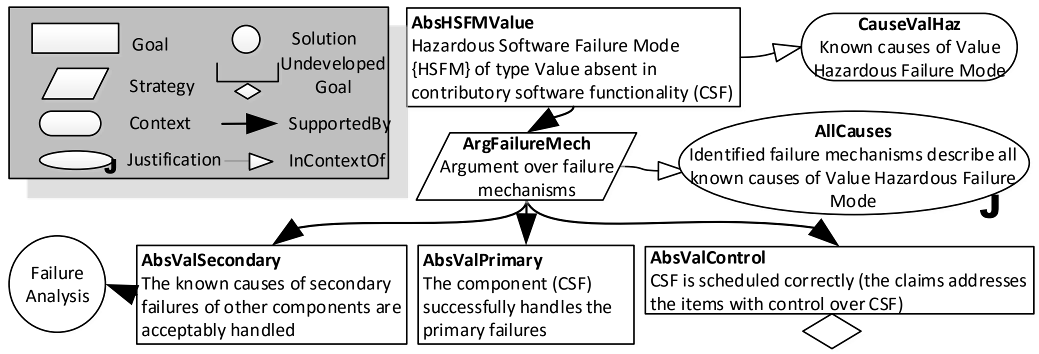

Goal
Solution
Undeveloped Goal
Strategy
Context
SupportedBy
Justification
InContextOf
AbsHSFMValue
Hazardous Software Failure Mode {HSFM} of type Value absent in contributory software functionality (CSF)
CauseValHaz
Known causes of Value Hazardous Failure Mode
ArgFailureMech
Argument over failure mechanisms
AllCauses
Identified failure mechanisms describe all known causes of Value Hazardous Failure Mode
Failure Analysis
AbsValSecondary
The known causes of secondary failures of other components are acceptably handled
AbsValPrimary
The component (CSF) successfully handles the primary failures
AbsValControl
CSF is scheduled correctly (the claims addresses the items with control over CSF)

Figure 6: Hazardous Software Failure Mode absence pattern for type value failure

tem is acceptably safe to operate in a given context [3]. A safety case is composed of all the work products produced during the development of a safety-critical system, which includes a safety argument that connects the safety requirements and the evidence supporting and justifying those requirements. The Goal Structuring Notation (GSN) is a graphical argumentation notation for representing the safety case [13]. GSN can be used to represent the individual elements of any safety argument and the relationships between these elements. The argument usually starts with a top-level claim/goal stating absence of a failure, as in Figure 6 the argument starts with a goal identified by *AbsHSFMValue*. The goals can be further decomposed to sub-goals with *supportedBy* relations denoting inference between goals or connecting supporting evidence with a goal. The decomposition can be described using strategy elements e.g., *ArgFailureMech* in Figure 6. To define the scope and context of a goal or provide its rationale, elements such as context and justification are attached to a goal with *inContextOf* relations. For example, context *CauseValHaz* is used to clarify the *AbsHSFMValue* goal by providing the list of known causes of the value failure mode. The *AllCauses* justification is used to justify why the *ArgFailureMech* strategy is sufficient to address the *AbsHSFMValue* top goal. The undeveloped element symbol indicates elements that need further development. For more details on GSN see [13].

GSN was initially used to communicate a specific argument for a particular system. Since similar rationale exists behind specific argument-fragments in different contexts, argument patterns of reusable reasoning are defined by generalising the specific details of a specific argument [13]. In this work we use the argument pattern for Handling of Software Failure Modes (HSFM) [10], a portion of which is shown in Figure 6, to structure the generated argument-fragments related to value failure modes. To build an argument, the HSFM pattern requires information about known causes of the failure mode and failure mechanisms that address those causes. Moreover, the failure mechanisms can be classified into three categories: (1) Primary failures within a Contributory Software Functionality (CSF) that can cause the failure; (2) Secondary failures relating to other components within the system on which the CSF is dependent; and (3) Failures caused by items controlling the CSF e.g., the scheduler.

2.4. ISO 26262

ISO 26262 [4] is a functional safety standard for the automotive domain. The current version of the standard is aimed at passenger vehicles up to 3500 kilograms. Despite that, industries that produce heavy vehicles, such as trucks and construction vehicles, are making efforts in aligning their development processes with ISO 26262 since future versions of the standard are planned to cover heavy vehicles as well [14, 15].

The ISO 26262 safety standard has been developed as a guidance to provide assurance that any unreasonable residual risks from malfunctioning E/E systems have been avoided. It explicitly requires a safety case in form of a clear and comprehensible argument to be built. Moreover, it sets the verification objectives that should be provided after each phase of the development process. The verification means includes combination of reviews, analyses and tests. Furthermore, the standard requires tool qualification for any tool that eliminates, reduces or automates any of the processes prescribed by the standard. For example, ISO 26262 requires software integration tests to be applied to demonstrate that the software components comply with the software architectural design (ISO 262626-6:2011, Clause 10) [16]. Moreover, the standard requires structural coverage analysis of the tests to evaluate their completeness. In case a tool for automated testing is used, an additional evidence should be provided to establish the tool qualification. While the software architectural design qualifies as the immediate evidence, the results of its testing are the direct evidence, while the tool qualification evidence of the tool used for automated testing is regarded as indirect evidence [17].

As mentioned in Section 1, the standard provides support for reuse of safety components developed out-of-context through the notion of Safety Element out-of-context (SEooC). SEooC is a notion that has been developed specifically for reuse according to ISO 26262. Besides SEooC, the standard supports two more reuse scenarios: (1) pre-existing elements not necessarily developed for reuse or according to ISO 26262 that have to be qualified for integration, and (2) elements that qualify for reuse as proven-in-use. In this work we focus on the SEooC reuse scenario.

2.5. Assurance case meta-model

Assurance case is a generic term for any case where an argument is used to connect the requirements with the supporting evidence (e.g., a safety or a security case). An assurance case is defined as *"a collection of auditable claims, arguments, and evidence created to support the contention that a defined system/service will satisfy the particular requirements."* [18]. Structured Assurance Case Meta-model (SACM) is an Object Management Group (OMG) standard that specifies a meta-model for representing structured assurance cases [18]. The purpose of the standardised meta-model is to provide better portability and exchange of the safety arguments used to represent the assurance cases. SACM consists of an argumentation meta-model and an evidence meta-model. The meta-model defines the assurance case as a composition of the argumentation captured by the argumentation meta-model and the evidence captured by the evidence meta-model.

Figure 7 shows the SACM argumentation meta-model (elements represented with solid borders). According to this meta-model, argumentation can be either an atomic argumentation, argument element, or a composition of different argumentations and/or argument elements. The argument element metaclass is a generalisation of citation elements for both evidence and other argument elements, and reasoning elements that include argument reasoning and assertions. While the argument reasoning captures the structure of the argumentation, the assertions metaclass represents generalisation of claims and the relationship between different argument elements. 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 [13] and Claims-Arguments-Evidence (CAE) [20]. 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.

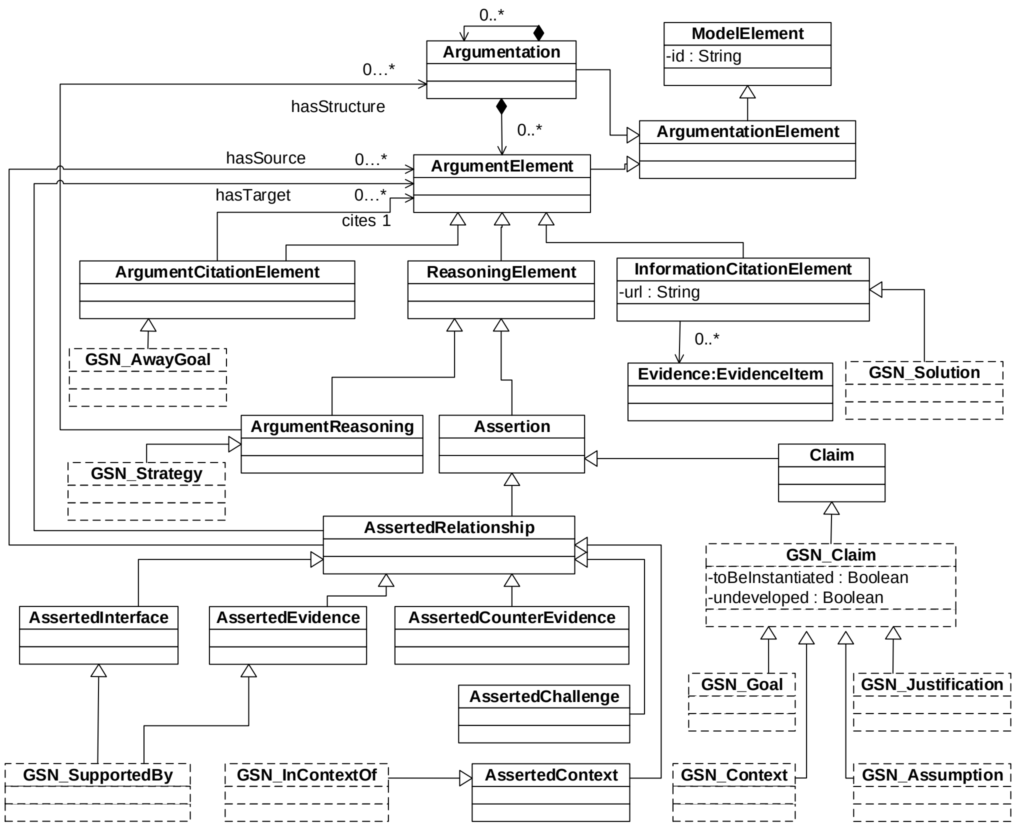


Figure 7: SACM argumentation meta-model with a subset of GSN elements (dashed borders) [19]

2.5.1. GSN meta-model

On top of the existing SACM argumentation meta-model elements, Figure 7 shows the meta-model extension that includes a subset of GSN elements (represented with dashed borders). The meta-model asserted relationships include the GSN inContextOf and supportedBy relationships, while the meta-model claim element includes the different GSN claims in form of propositions such as goals, justifications, assumptions and contexts. The information citation element of the meta-model is a generalisation of the GSN solution element, while the argument citation element is a generalisation of the GSN away goal element.

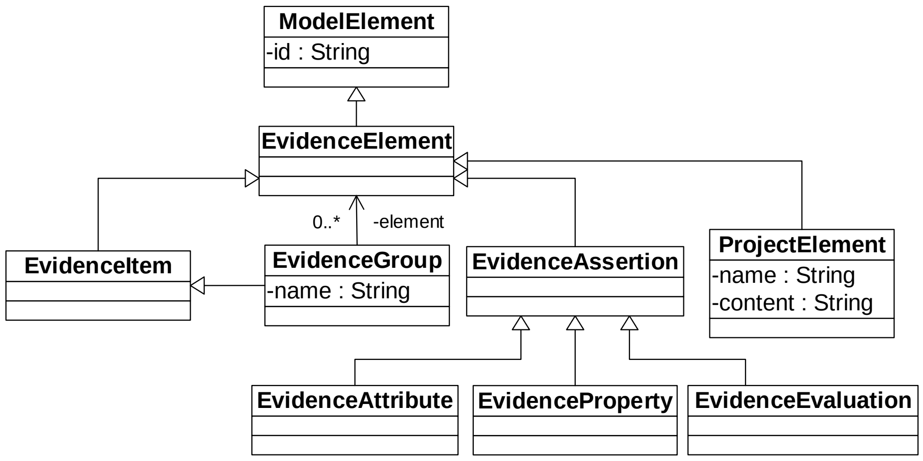


Figure 8: A part of SACM evidence meta-model [18]

2.5.2. Evidence meta-model

As mentioned in Section 2.3, the evidence is one of the main pillars of safety cases alongside the requirements they support and the argument which connects the two. *Evidence* is information or an objective artefact offered in support of one or more claims [18]. 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 [18], or based on the characteristics of the document that is the source of the information [17]. We focus on the categorisation of evidence based on the nature of support it offers [17], i.e., proximity of the evidence to the product it supports, which categorises evidence as immediate, direct and indirect evidence. The *immediate evidence* represents the original artefacts that is being evaluated as evidence such as source code, 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/personnel qualifications and development process.

While the SACM argumentation meta-model can be used as a standalone specification, it can be used in combination with the evidence meta-model that provides additional support for evidence management within assurance cases. Figure 8 presents a portion of the SACM evidence meta-model [18]. The main logical parts of the evidence meta-model are the evidence items and evidence assertions. The evidence item defines the physical evidence such as those provided in form of documents, while the evidence assertion defines various statements about the essential properties of evidence items. Some of the main groups of such statements include evidence evaluations, evidentiary support and statements related to the fundamental properties of the evidence independent of the particular assurance case. The evidence evaluations include statements about the relationship between the evidence items and the argumentation claims. The evidentiary support statements are made on the nature of support that evidence items confer on the claims they support (e.g., direct or indirect support). The fundamental properties of the evidence that are independent of particular assurance case include information such as the author of the statement, media of a document, and the current custodian of the document.

3. Safety Element out-of-context Meta-Model

In this section we first present the Safety Element out-of-context Meta-Model (SEooCMM) and then we discuss its relationship with the SACM evidence meta-model. Finally, we present how a SEooCMM compliant source model can be transformed into a SACM compliant target model.

3.1. SEooCMM

The SEooCMM (Figure 9) represents an extension of the component meta-model presented in Section 2.1. It adds support for modelling the basic argumentation elements (undeveloped element, context, justification) and provides standardised support for evidence management. The *Safety Contract* metaclass is 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. The meta-model does not provide explicit support for modelling argumentation assumptions, but all assumptions should be captured within the contract assumptions.

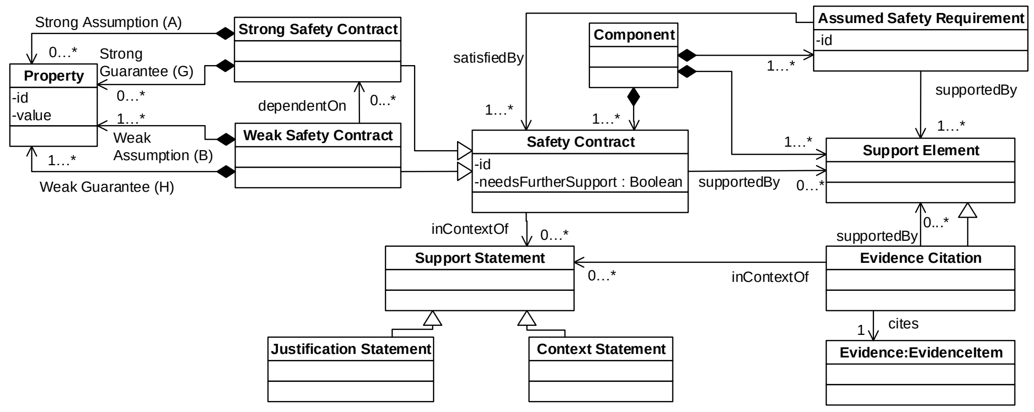


Figure 9: SEooCMM extension with the connecting elements to SACM argumentation and evidence meta-models

The main extension to the component meta-model is support for fine-grained modelling of the supporting elements and statements. The abstract metaclass *Support Element* is used for modelling the evidence items that can be used to support the contracts and the related assumed safety requirements. Evidence management in SEooCMM is supported by establishing a connection with the SACM evidence meta-model. The *Evidence Citation* metaclass refers to a single *EvidenceItem* metaclass from the SACM evidence meta-model, which establishes the support relationship between the evidence elements in the SACM evidence meta-model and the safety contracts and requirements in SEooCMM. Explicit support for specification of the GSN argumentation Context and Justification elements (described in Section 2.3) is achieved through the *Support Statements* metaclass, i.e., its sub-metaclasses *Context Statement* and *Justification Statement*.

Table 1 represents an example of a SEooCMM compliant specification for the running example introduced in Section 1. The Estimator component is described by the allocated safety requirement *SR1*, $\langle A, G\rangle_{Estimator-1}$ contract, and the $\mathrm{E}_{Estimator-1}$ evidence citation. The specified contract is clarified with the context statement $\mathrm{C}_{Estimator-1}$, while the evidence citation $\mathrm{E}_{Estimator-1}$ is described by the statement $\mathrm{E}_{Estimator-1}(desc)$. Furthermore, the evidence $\mathrm{E}_{Estimator-1}$ is further supported by a tool qualification argument-fragment $\mathrm{E}_{Estimator-1}(supporting\ argument)$. The $\mathrm{NFS}_{Estimator-1}$ flag indicates that the contract is fully validated. The *needsFurtherSupport* flag is false by default, unless specified otherwise.

Table 1: An example of a SEooCMM compliant specification

SR1:	The *estimatedSensorValue* shall be normalised with the maximum error margin ±5%;
$\mathbf{A}_{Estimator-1}$: $\mathbf{G}_{Estimator-1}$: $\mathbf{NFS}_{Estimator-1}$:	*sensorValue* within [0, 5] AND *parameter1* within [230, 1000] AND *sensorValue* error margin within ±10% AND *sensorValue* consecutive value failures less than 3; *estimatedSensorValue* error margin within ±2%; false;
$\mathbf{C}_{Estimator-1}$:	The error margin established with respect to an ideal sensor;

$\mathbf{E}_{Estimator-1}$:	*name*: Estimator Simulation Results;
	desc: Simulation performed under the assumed conditions;
	supporting argument: Simulator_qualifications_arg;

3.2. Evidence management in SEooCMM

The connection between the three meta-models is shown in Figure 10. As mentioned in Section 2.5, the main benefit of support for SACM compliant modelling is the standardised format which facilitates portability. The portability is an important aspect for reuse. Reuse of software components is not sufficient without reuse of the "other knowledge" [21], which has particular significance for safety-critical systems where the safety-relevant artefacts about the component can incur significant costs as mentioned in Section 1. To support portability of evidence items within SEooCMM we use the *Evidence Container* compliant to the SACM evidence meta-model for capturing the information related to the evidence. We use automated transformations to generate GSN argumentation from a SEooCMM compliant specification. Since SEooCMM captures information out-of-context, the transformation of the SEooCMM compliant specification results in a set of different GSN argument-fragments, depending on the context for which the transformation is performed. Moreover, the generated argument-fragments can be combined in different ways to compile GSN arguments with different architectures. Since both the GSN argumentation model and the SEooCMM compliant model use evidence container compliant to the same SACM evidence meta-model, portability of evidence together with the generated argumentation is made easier.

To capture the nature of the evidentiary support of the contracts, we enrich the connection between a safety contract and evidence by relating the SEooCMM to the SACM evidence meta-model. As mentioned in Sec-

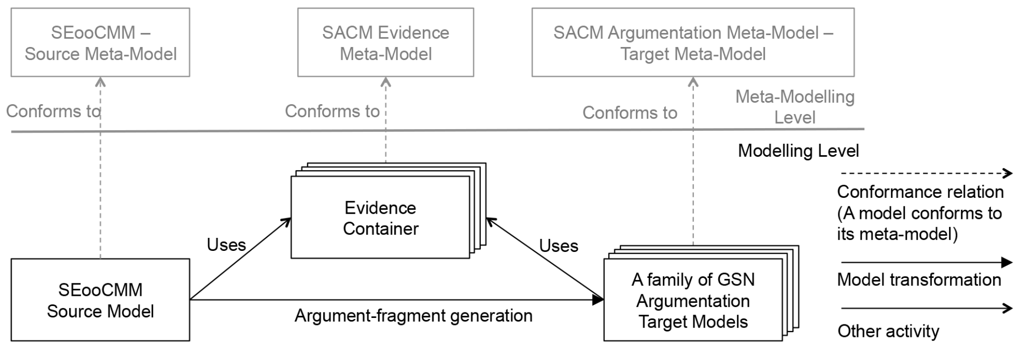

SEooCMM – Source Meta-Model
SACM Evidence Meta-Model
SACM Argumentation Meta-Model – Target Meta-Model
Conforms to
Conforms to
Conforms to
Meta-Modelling Level
Modelling Level
Evidence Container
Uses
Uses
SEooCMM Source Model
A family of GSN Argumentation Target Models
Argument-fragment generation
Conformance relation (A model conforms to its meta-model)
Model transformation
Other activity

Figure 10: Connection between the SEooCMM and SACM meta-models

tion 2.5.2, 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 evidence. The nature of the support of such evidence to the contracts can be further detailed to distinguish between evidence that supports contract consistency, completeness and correctness [9].

SEooCMM allows evidence items to be supported by other supporting elements including other evidence items such as indirect evidence (e.g., $\mathrm{E}_{Estimator-1}(\textit{supporting argument})$ in Table 1), and supporting statements about the evidence (e.g., $\mathrm{E}_{Estimator-1}(desc)$). The captured evidence-related information is used to provide additional clarification of the connection between the evidence and the claims generated from the contracts that are supported by this evidence. Clarification of confidence in the evidence itself can be made in two different ways: either by supporting the direct evidence with another evidence; or by using the supporting statements to clarify or justify why this evidence is sufficient to address a particular contract. Supporting an evidence item with other evidence can be done either by directly relating direct evidence with an indirect evidence (e.g., competence of person performing the failure analysis can be found in document x); or by pointing to an already developed goal, called an *away goal* [13], which presents the indirect evidence and the supporting information (we could have a repository of generic argument-fragments related to staff competence and tool-qualification [22]). SEooCMM, through SACM evidence meta-model, supports capturing of attributes of the evidence items in form of evidence assertions, while the evidentiary support that the evidence items provide to the safety contracts is established through the evidence evaluations elements.

Algorithm 1 M2M Transformation from SEooCMM to GSN SACM argumentation meta-model

```
SEooCMM2SACM(in SEooCMM, out SACM){
  for each satisfied SafetyContract sc in SEooCMM do
    sc2claim(in SEooCMM::SafetyContract, out SACM::GSN_Goal);
    scCont(in SEooCMM::Context, out SACM::GSN_Context);
    scJust(in SEooCMM::Justification, out SACM::GSN_Justification);
    addSubGoals(in SEooCMM::SafetyContract, out SACM::GSN_Goal);
    sc2context(in SEooCMM::SafetyContract, out SACM:: GSN_Context);
    for each satisfied Strong/Weak Assumption a in sc do
      a2claim(in SEooCMM::SafetyContract, out SACM::GSN_Goal);
      away2a (in SEooCMM::SafetyContract, out SACM::GSN_AwayGoal);
    end for
    for each EvidenceCitation ec supporting sc do
      ec2claim(in SEooCMM::EvidenceCitation, out SACM::GSN_Goal);
      ecSol(in Evidence::EvidenceItem, out SACM::GSN_Solution);
      ecCont(in SEooCMM::Context, out SACM::GSN_Context);
      ecJust(in SEooCMM::Justification, out SACM::GSN_Justification);
    end for
  end for
}
```

3.3. SEooCMM to SACM argumentation meta-model transformation

Transforming the information captured within a model compliant with SEooCMM to a GSN argumentation model compliant with the SACM argumentation meta-model results in a set of argumentation-fragments about contract satisfaction that can be organised in different ways, for instance to argue that the safety requirements allocated to the component have been satisfied by the contracts [9]. Regardless of the organisation of the generated argument-fragments, there are certain transformation rules that are common for generation of all the resulting argument-fragments.

The common transformation rules are summarised by means of pseudocode in Algorithm 1. The *SEooCMM2SACM* model-to-model (M2M) transformation generates a set of argument-fragments, one for each satisfied safety contract. The algorithm is composed of three main steps:

- **Step 1** – The guarantees of the contract are transformed to claims in *sc2claim* and further decomposed in *addSubGoals* onto the two subgoals to separate goals related to assumptions satisfaction and evidential support. The initial goal is further clarified with context statements regarding the originating safety contract in *sc2context*, while any supporting elements are associated with the goal in *scCont* and *scJust*.

- **Step 2** – 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** – For each evidence element associated with a contract a goal is created from the evidence citation element in *ec2claim*, and then supported with solutions in *ecSol*, and with other supporting elements in *ecCont* and *ecJust*.

4. FLAR2SAF

In this section we present FLAR2SAF, a method to generate reusable safety case argument-fragments. We first provide the rationale of the approach in Section 4.1. We provide a method to translate FPTC rules into safety contracts in Section 4.2, and we adapt and extend the method for semi-automatic generation of argument-fragments from safety contracts in Section 4.3.

4.1. Rationale

In our work we use safety contracts to facilitate reuse of safety-relevant software components. The method to semi-automatically generate argument-fragments from safety contracts, mentioned in Section 2.1, can be used to support the reuse of certification-relevant artefacts from previously specified contracts. Just as direct evidence needs to be provided with a reusable component to increase confidence in the component itself, in some cases indirect evidence needs to be provided to increase trustworthiness of the direct evidence [23]. To reuse evidence-related artefacts together with the argument fragments, SEooCMM captures the additional information about linking the artefacts and the safety contracts. Furthermore, SEooCMM addresses the issue of trustworthiness of such evidence by allowing evidence items to be supported by other evidence and supporting statements. For example, in case we need to describe the competence of the engineers that performed a particular analysis or qualification of the analysis tool, the analysis results evidence item can be supported by the indirect evidence about the personnel/tool qualification.

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 enrich the contract assumptions accordingly;
5. Upon component selection and satisfaction of the strong safety contracts, depicted in Figure 3 in Section 2.1:
   (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 approach to semi-automatically generate an argument-fragment based on the argument pattern presented in Section 2.3 (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.

4.2. 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 2) to explain the translation process and provide a set of steps that can be used to perform the translation. In Table 2 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

Table 2: FPTC rules of the Estimator, Converter, and Filter components

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:	$sV.omission, p1.wildcard \rightarrow eSV.omission$;
	$sV.wildcard, p1.omission \rightarrow eSV.omission$;
	$sV.valueCoarse, p1.noFailure \rightarrow eSV.noFailure$;
	$sV.noFailure, p1.valueCoarse \rightarrow eSV.noFailure$;
	$sV.valueCoarse, p1.valueCoarse \rightarrow eSV.noFailure$;

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.

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 3, 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

Table 3: The translated contract examples for the Estimator component

$\mathbf{A}_{Estimator-1}$: {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;

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 3 for the $\langle A, G\rangle_{Estimator-2}$ contract.

The third type of safety contracts that we translate from FPTC rules are 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 3 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 2.2, 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 [11], 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 on the example of translating FPTC rules from Table 2 to contracts in Table 3, the translation can be performed in the following way for each failure:

- 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 describing when the failure mode happens:
  - Add the negation of the combination of the input failures to the contract assumptions. Connect with other assumptions with AND operator;
  - Use the absence of the failure mode as the contract guarantee;
- For the rules that describe behaviours that prevent the failure mode:
  - Use the rule within the contract guarantee to state that the component guarantees the behaviour described by the rule;
- For the third type of contracts:
  - 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;

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.

4.3. Argument-fragment generation

As mentioned in Section 2, safety relevant components usually need to provide argument and associated evidence regarding absence of particular failures. We generate the required argument-fragment based on an already established argument pattern for presenting absence of value failure mode, briefly recalled in Section 2.3. By providing means to generate context-specific argument-fragments, i.e., argument-fragments that include only information related to those contracts satisfied in the particular context, we allow for reuse of certain evidence related to the satisfied contracts.

To build an argument based on the HSFM pattern, we identify the known causes of primary and secondary failures from the corresponding FPTC rules. We identify the primary failures from the contracts translated from FPTC rules that describe behaviours that mitigate a failure mode. The secondary failures are captured within the contracts translated from FPTC rules that describe when a failure mode happens. All causes and assumptions not captured by the corresponding FPTC rules should 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 2.3. We adapt the top-level argument-fragment from [9] to further develop the sub-goals.

We use the safety contracts to generate the supporting sub-arguments for the primary and secondary failures and leave the goal related to controlling failures undeveloped. Supporting sub-arguments for both primary and secondary failures are composed from the argument-fragments generated from the satisfied safety contracts. The argument-fragments for each of the contracts argues that the corresponding safety contract is satisfied with sufficient confidence. The sufficient confidence is determined based on the specific SIL of the requirements allocated on the component and may require additional evidence in case of higher SILs. We use the transformation rules presented in Section 3.3 for the generation of the argument-fragments for satisfaction of the contract, where we make a claim that the contract is satisfied with sufficient confidence, i.e., that the guarantee of the contract is offered. We further decompose the claim into two supporting goals:

- an argument providing the supporting evidence for confidence in the claim in terms of completeness of the contract, and

- an argument showing that the assumptions stated in the contract are met by the contracts of other components.

We further focus on the first sub-goal related to evidence which includes the additionally specified information about the evidence artefacts.

For every evidence attached to a safety contract we create a sub-goal to support confidence in the corresponding safety contract. At this point we can use the additional information about the rationale connecting evidence and the safety contract and present it in form of a context statement to clarify how this particular evidence contributes to increasing confidence in the corresponding safety contract. The evidence can be further backed up by the related trustworthiness arguments that can be attached directly to a particular evidence. If the evidence trustworthiness information is provided in a descriptive form then additional context statements are added to the solutions, otherwise an away goal is created to point to the argument about the trustworthiness of the evidence, e.g., an argument presenting competence of a person that conducted the analysis which resulted in the corresponding evidence.

To achieve the argument-fragment generation we extended the approach for generation of argument-fragments from safety contracts [9] to allow for argument-fragment generation in the specific form of the selected pattern. While the core of the generation are the argument-fragments for each of the contracts, the way these argument-fragments are organised into a larger argument can differ. The approach is adapted to generate an argument-fragment that clearly separates and argues over primary, secondary and controlling failures as described above, and to include additional information related to the evidence.

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

5. Case Study

In this section we first briefly present the case study methodology in Section 5.1, and then introduce the case of study in Section 5.2. In Section 5.3 we apply CHESS-FLA/FPTC analysis on a reusable component and use the translation steps from Section 4.2 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 5.4. We present the generated argument-fragment in Section 5.5. In Section 5.6 we provide a discussion and then examine the case study validity in Section 5.7.

5.1. Case Study Methodology

Case study is an empirical method for investigating a contemporary phenomenon in its real-world context [24]. Software engineering research often relies on case study methodology for different purposes. For example, exploratory case studies are used to generate new ideas, while explanatory case studies are used to seek for a solution to a problem. Regardless of the purpose of a case study, its crucial part is *the case* [25]. The case is the object of study and should be a sufficiently complex component investigated in its natural and real-world context.

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 within the construction vehicles domain. More specifically, we conduct an explanatory case study to answer the following research questions:

- **RQ1**: *Can FPTC analysis be performed if inputs of all components under analysis do not consider the same set of failures?*
- **RQ2**: *Is reuse of FPTC-related safety artefacts achievable 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 a 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, i.e. heavy equipment machines used in construction to move/load material onto/into other

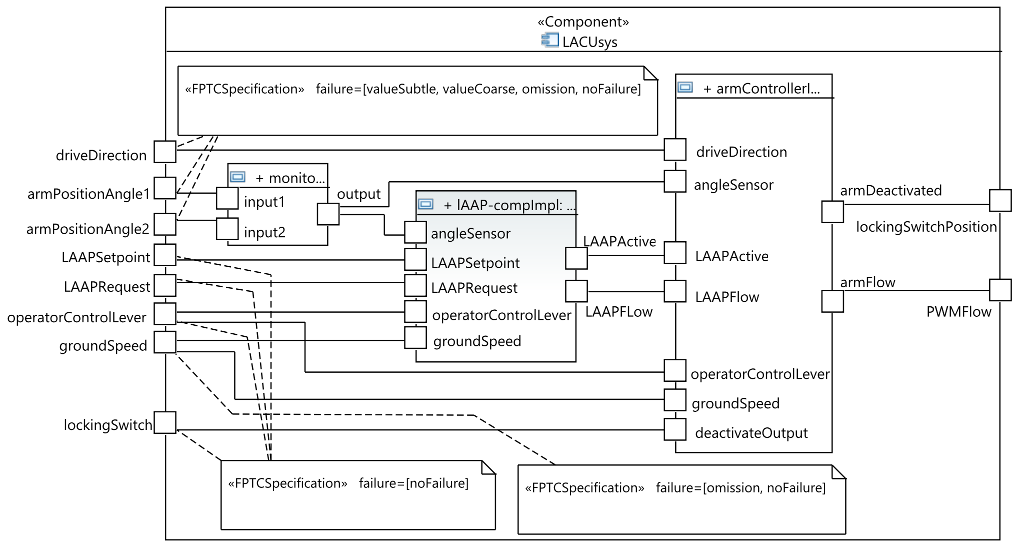


Figure 11: LACU model in CHESS

types of vehicles. 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.

5.2. The Case

Wheel-loaders are usually equipped with a loading arm, which can perform up and down movements. In this case we are focusing on the development of a Compact Wheel-loader (CWL), which is often used for tasks that require high precision of the arm movement. Moreover, CWL is not used only in construction sites, but often for public service in areas with pedestrians.

LACU is a software control unit that based on sensory data and user input calculates the arm movement commands and issues them to a hydraulic controller that moves the arm physically. The software architecture of LACU modelled in CHESS is shown in Figure 11. LACU is composed of three subcomponents: the Monitor component that keeps track of the dual angle sensor of the loading arm; the LAAP component that handles automatic arm positioning; and the ArmController component that issues the final arm positioning command. The LAAP component is developed independently of this system as an out-of-context reusable component [26].

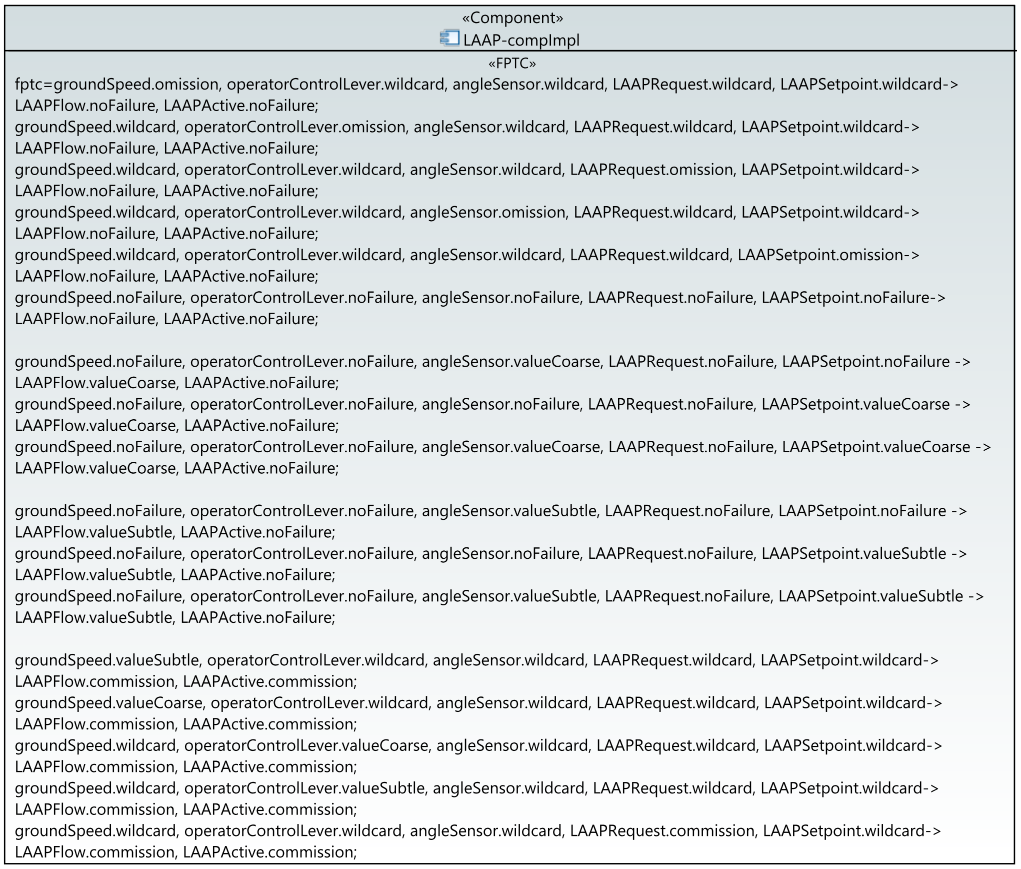
The hazard analysis of the loading arm has identified a vehicle level hazard *H1: unintended movement of the lifting arm*, which can be dangerous in different operational situations in which CWL is used. Angle sensor value failure is identified as a contributor to the hazard H1. As one of the safety 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 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 CWL. In the next section we will focus on the FPTC analysis of LAAP and present its contracts and the accompanying evidence.

5.3. LAAP Failure Logic Analysis

The LAAP component is highlighted in the LACU architecture in Figure 11. 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-context development [26], and detail the translated contracts.

The FPTC rules representing the LAAP failure behaviour are shown in Figure 12. 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,



```
«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 12: A subset of LAAP FPTC rules

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 commission, 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

Table 4: A subset of the translated LAAP strong contracts with the associated evidence

$\mathbf{A}_{LAAP-1}$: $\mathbf{G}_{LAAP-1}$:	{*groundSpeed*, *operatorControlLever*, *angleSensor*, *LAAPSetpoint*}.failure within {omission, valueSubtle, valueCoarse} AND *LAAPRequest.failure* within {omission, commission}; *LAAPFlow*.failure within {valueSubtle, valueCoarse, commission} AND *LAAPActive*.failure within {commission} AND not *LAAPFlow.omission* AND not *LAAPActive.omission*;
$\mathbf{C}_{LAAP-1}$:	The contract is derived from the FPTC analysis results for the LAAP component;
$\mathbf{E}_{LAAP-1}$:	*name*: LAAP FPTC analysis report *description*: FPTC analysis is performed in CHESS-toolset. *supporting argument*: FPTC_rules_conf;
$\mathbf{A}_{LAAP-2}$: $\mathbf{G}_{LAAP-2}$:	-; groundSpeed.omission, operatorControlLever.wildcard, angleSensor.wildcard, LAAPRequest.wildcard, LAAPSetpoint.wildcard $\rightarrow$ LAAPFlow.noFailure, LAAPActive.noFailure;
$\mathbf{C}_{LAAP-2}$:	The contract is derived from the FPTC analysis results for the LAAP component; Unit testing is used to validate that the contracts are sufficiently complete with respect to the implementation;
$\mathbf{E}_{LAAP-2}$:	*name*: LAAP FPTC analysis report *description*: FPTC analysis is performed in CHESS-toolset. *supporting argument*: FPTC_rules_conf; *name*: Unit testing results *description*: - *supporting argument*: Unit_test_conf;

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 4.2. The translated strong contracts are shown in Table 4. 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 $\langle A, G\rangle_{LAAP-1}$ imposes restrictions on the environment of the component by making assumptions that the component can receive on its input ports only those failures con-

Table 5: The translated LAAP weak contracts with the associated evidence

$\mathbf{B}_{LAAP-1}$: $\mathbf{H}_{LAAP-1}$:	not *angleSensor.valueCoarse* AND not *LAAPSetpoint.valueCoarse*; not *LAAPFlow.valueCoarse*;
$\mathbf{C}_{LAAP-1}$:	The contract is derived from the FPTC analysis results for the LAAP component;
$\mathbf{E}_{LAAP-1}$:	*name*: LAAP FPTC analysis report *description*: FPTC analysis is performed in CHESS-toolset. *supporting argument*: FPTC_rules_conf;
$\mathbf{B}_{LAAP-2}$: $\mathbf{H}_{LAAP-2}$:	not *angleSensor.valueSubtle* AND not *LAAPSetpoint.valueSubtle*; not *LAAPFlow.valueSubtle*;
$\mathbf{C}_{LAAP-2}$:	The contract is derived from the FPTC analysis results for the LAAP component;
$\mathbf{E}_{LAAP-2}$:	*name*: LAAP FPTC analysis report *description*: FPTC analysis is performed in CHESS-toolset. *supporting argument*: FPTC_rules_conf;
$\mathbf{B}_{LAAP-3}$: $\mathbf{H}_{LAAP-3}$:	not *groundSpeed.valueSubtle* AND not *groundSpeed.valueCoarse* AND not *operatorControlLever.valueSubtle* AND not *operatorControlLever.valueCoarse* AND not *LAAPRequest.commission*; not *LAAPFlow.commission* AND not *LAAPActive.commission*;
$\mathbf{C}_{LAAP-3}$:	The contract is derived from the FPTC analysis results for the LAAP component;
$\mathbf{E}_{LAAP-3}$:	*name*: LAAP FPTC analysis report *description*: FPTC analysis is performed in CHESS-toolset. *supporting argument*: FPTC_rules_conf;

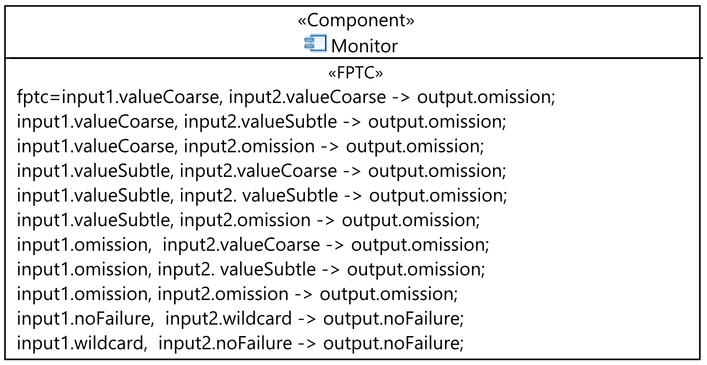
sidered within the FPTC analysis for this component. More specifically, the component considers omission and commission on LAAPRequest, and omission, valueSubtle and valueCoarse on other input ports. The strong contract then indicates that the component guarantees that it will not exhibit omission failures, 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 contracts 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 5. The FPTC rules about the valueCoarse failure of the LAAPFlow port combined are translated to the contract $\langle B, H\rangle_{LAAP-1}$. 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 $\langle B, H\rangle_{LAAP-2}$ 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 $\langle B, H\rangle_{LAAP-3}$ indicates that to prevent commission of both of the outputs, there should be no value failures on groundSpeed and operatorControlLever 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.

5.4. LACU Failure Logic Analysis

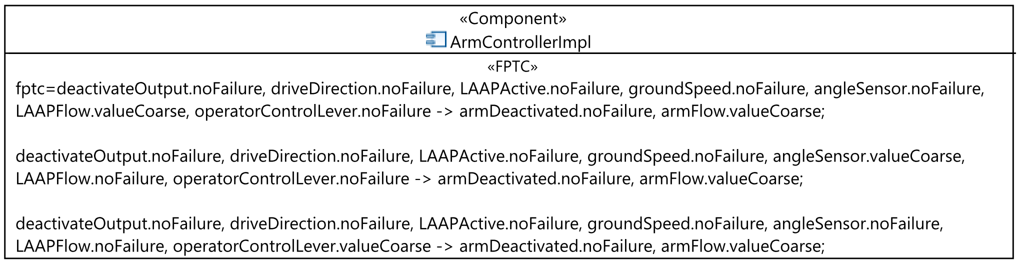
As mentioned in the Section 5.2, the LACU hazard analysis indicates that the value failures of the angle sensor can lead to the hazard H1. Hence, when selecting the subcomponents for this system, their failure behaviour related to the value failures needs to be investigated to ensure that value failures 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 $\langle A, G\rangle_{LAAP-1}$ 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 $\langle B, H\rangle_{LAAP-1}$



«Component»
Monitor
«FPTC»
fptc=input1.valueCoarse, input2.valueCoarse -> output.omission;
input1.valueCoarse, input2.valueSubtle -> output.omission;
input1.valueCoarse, input2.omission -> output.omission;
input1.valueSubtle, input2.valueCoarse -> output.omission;
input1.valueSubtle, input2. valueSubtle -> output.omission;
input1.valueSubtle, input2.omission -> output.omission;
input1.omission, input2.valueCoarse -> output.omission;
input1.omission, input2. valueSubtle -> output.omission;
input1.omission, input2.omission -> output.omission;
input1.noFailure, input2.wildcard -> output.noFailure;
input1.wildcard, input2.noFailure -> output.noFailure;

Figure 13: The set of Monitor FPTC rules



«Component»
ArmControllerImpl
«FPTC»
fptc=deactivateOutput.noFailure, driveDirection.noFailure, LAAPActive.noFailure, groundSpeed.noFailure, angleSensor.noFailure, LAAPFlow.valueCoarse, operatorControlLever.noFailure -> armDeactivated.noFailure, armFlow.valueCoarse;

deactivateOutput.noFailure, driveDirection.noFailure, LAAPActive.noFailure, groundSpeed.noFailure, angleSensor.valueCoarse, LAAPFlow.noFailure, operatorControlLever.noFailure -> armDeactivated.noFailure, armFlow.valueCoarse;

deactivateOutput.noFailure, driveDirection.noFailure, LAAPActive.noFailure, groundSpeed.noFailure, angleSensor.noFailure, LAAPFlow.noFailure, operatorControlLever.valueCoarse -> armDeactivated.noFailure, armFlow.valueCoarse;

Figure 14: A subset of the ArmController FPTC rules

and $\langle B, H\rangle_{LAAP-2}$ 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 5.2, the software monitor and the error-detecting code have been implemented to ensure that the contracts $\langle B, H\rangle_{LAAP-1}$ and $\langle B, H\rangle_{LAAP-2}$ are satisfied.

The Monitor FPTC rules are shown in Figure 13. 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 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 14.

To perform the FPTC analysis on the LACU modelled in the CHESS-

Table 6: A subset of the translated LACU strong contracts with the associated evidence

$\mathbf{A}_{LACU-1}$: $\mathbf{G}_{LACU-1}$:	*lockingSwitch* noFailure AND {*driveDirection*, *groundSpeed*}.failure within {omission, commission, valueSubtle, valueCoarse} AND {*armPositionAngle1*, *armPositionAngle2*, *LAAPSetpoint*, *operatorControlLever*}.failure within {omission, valueSubtle, valueCoarse} AND *LAAPRequest*.failure within {omission, commission}; {*PWMFlow*, *lockingSwitchPosition*}.failure within{commission} AND not $PWMFlow.valueCoarse$ AND not $PWMFlow.valueSubtle$ AND not $PWMFlow.omission$ AND not *lockingSwitchPosition.omission*;
$\mathbf{C}_{LACU-1}$:	The contract is derived from the FPTC analysis results for the LACU component;
$\mathbf{E}_{LACU-1}$:	*name*: LACU FPTC analysis report *description*: FPTC analysis is performed in CHESS-toolset. *supporting argument*: FPTC_rules_conf;
$\mathbf{A}_{LACU-2}$: $\mathbf{G}_{LACU-2}$:	-; lockingSwitch.noFailure, driveDirection.noFailure, armPositionAngle1.valueCoarse, armPositionAngle2.noFailure, LAAPSetpoint.noFailure, LAAPRequest.noFailure, groundSpeed.noFailure, operatorControlLever.noFailure $\rightarrow$ lockingSwitchPosition.noFailure, PWMFlow.noFailure
$\mathbf{C}_{LAAP-2}$:	Unit testing is used to validate that the contracts are sufficiently complete with respect to the implementation;
$\mathbf{E}_{LAAP-2}$:	*name*: Unit testing results *description*: - *supporting argument*: Unit_test_conf;

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 11, 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

Table 7: A translated LACU weak contract with the associated evidence

$\mathbf{B}_{LACU-1}$:	not *lockingSwitch.commission* AND not *groundSpeed.valueSubtle* AND not *groundSpeed.valueCoarse* AND not *operatorControlLever.valueSubtle* AND not *operatorControlLever.valueCoarse* AND not *LAAPRequest.commission*;
$\mathbf{H}_{LACU-1}$:	not *PWMFlow.commission* AND not *lockingSwitchPosition.commission*;
$\mathbf{C}_{LACU-1}$:	The contract is derived from the FPTC analysis results for the LACU component;
$\mathbf{E}_{LACU-1}$:	*name*: LACU FPTC analysis report *description*: FPTC analysis is performed in CHESS-toolset. *supporting argument*: FPTC_rules_conf;

the component environment. For commission not to occur the environment needs to fulfil certain assumptions (Table 7), such as locking switch should not exhibit 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 6), 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 7).

5.5. The resulting argument-fragment

Based on the LACU contract specification compliant to SEooCMM we have applied the transformation rules presented in Section 3 to generate the argument-fragment that argues absence of value failure mode in the LACU. A part of the resulting argument-fragment is shown in Figure 15. In contrast to the total argument-fragment, the argument snippet for the $\langle A, G\rangle_{LACU-1}$ contract lacks only some of the away goals pointing to the contracts in the environment of LACU that satisfy the assumptions of $\langle A, G\rangle_{LACU-1}$.

The *AbsValPrimary* goal is supported by the satisfied contracts that are related to mitigation of value failures by the LACU, while the *AbsValSecondary* goal addresses the weak contracts that require the environment of LACU to ensure that the value failure mode for the LACU do not result in value failures. The strong contracts $\langle A, G\rangle_{LACU-1}$ and $\langle A, G\rangle_{LACU-2}$ are identified as contributing to mitigation of the primary causes of the value failures of LACU, hence their satisfaction is argued under the *ArgAbsValPrimary* strategy. By applying the rules to generate the contract satisfaction argument, we further develop the LACU-1 satisfaction goal with identifier "$\langle A, G\rangle LACU-1\_sat$", while we leave the goal "$\langle A, G\rangle LACU-2\_sat$" undeveloped. The LACU-1 satisfaction goal is divided to argue over the satisfaction of the supporting contracts and supporting evidence in contract completeness ("$\langle A, G\rangle LACU-1\_confidence$"). The supporting contracts of $\langle A, G\rangle_{LACU-1}$ include contracts supporting its assumptions and the supporting 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. While the argument for the "$\langle A, G\rangle LACU-1\_supp\_sat$" goal follows the same pattern as for goal "$\langle A, G\rangle LACU-1\_sat$", we focus on the argument related to the "$\langle A, G\rangle LACU-1\_confidence$" goal.

The goal "$\langle A, G\rangle LACU-1\_confidence$" is clarified by a context statement stating that the contract has been derived from the FPTC analysis. In the rest of the argument we create a goal for each of the attached artefacts and enrich them with additional evidence information. The goal "$\langle A, G\rangle LACU-1\_1$" presents the confidence in the FPTC analysis. Since we do not have an argument supporting qualification of the tool used to perform the analysis we attach context statement clarifying that the FPTC analysis is performed in the CHESS-toolset. We provide an away goal related to the evidence to support trustworthiness in the analysis by arguing confidence in the FPTC analysis. Further evidence might be provided to present competences of the engineers that formed the FPTC rules and performed the analysis.

Since value failures are handled by the LACU component, the *AbsValSecondary* goal remains undeveloped. Conversely, when generating an argument based on the HSFM pattern for commission failure, the *AbsValSecondary* goal would contain an argument over satisfaction of the $\langle B, H\rangle LACU-1$ contract. This contract indicates that LACU relies on its environment to contain certain causes of the commission failure in order to mitigate it.

5.6. 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

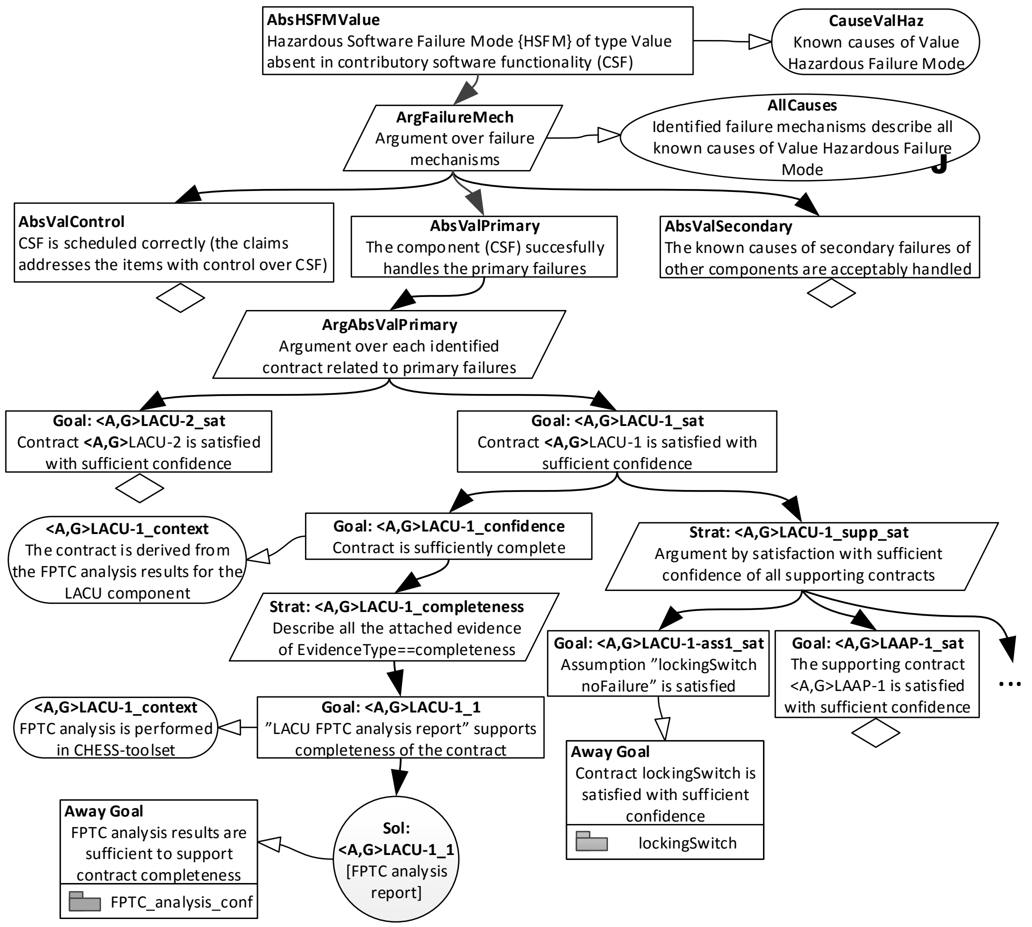

AbsHSFMValue
Hazardous Software Failure Mode {HSFM} of type Value absent in contributory software functionality (CSF)
CauseValHaz
Known causes of Value Hazardous Failure Mode
ArgFailureMech
Argument over failure mechanisms
AllCauses
Identified failure mechanisms describe all known causes of Value Hazardous Failure Mode
J
AbsValControl
CSF is scheduled correctly (the claims addresses the items with control over CSF)
AbsValPrimary
The component (CSF) succesfully handles the primary failures
AbsValSecondary
The known causes of secondary failures of other components are acceptably handled
ArgAbsValPrimary
Argument over each identified contract related to primary failures
Goal: <A,G>LACU-2_sat
Contract <A,G>LACU-2 is satisfied with sufficient confidence
Goal: <A,G>LACU-1_sat
Contract <A,G>LACU-1 is satisfied with sufficient confidence
<A,G>LACU-1_context
The contract is derived from the FPTC analysis results for the LACU component
Goal: <A,G>LACU-1_confidence
Contract is sufficiently complete
Strat: <A,G>LACU-1_supp_sat
Argument by satisfaction with sufficient confidence of all supporting contracts
Strat: <A,G>LACU-1_completeness
Describe all the attached evidence of EvidenceType==completeness
Goal: <A,G>LACU-1-ass1_sat
Assumption "lockingSwitch noFailure" is satisfied
Goal: <A,G>LAAP-1_sat
The supporting contract <A,G>LAAP-1 is satisfied with sufficient confidence
...
<A,G>LACU-1_context
FPTC analysis is performed in CHESS-toolset
Goal: <A,G>LACU-1_1
"LACU FPTC analysis report" supports completeness of the contract
Away Goal
Contract lockingSwitch is satisfied with sufficient confidence
lockingSwitch
Away Goal
FPTC analysis results are sufficient to support contract completeness
FPTC_analysis_conf
Sol:
<A,G>LACU-1_1
[FPTC analysis report]

Figure 15: Argument-fragment based on the HSFM pattern

input ports of a component, skipping some failure modes on the input ports of such components becomes inevitable [11]. 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 [27].

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 skipped FPTC rules [11], it would be useful to identify if the set of FPTC rules of the reusable component is 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 *RQ1* stated in Section 5.1 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* stated in Section 5.1 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 [9]. 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. In contrast to fully automated approaches, FLAR2SAF offers less automation and more manual effort is needed. 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 [28], 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 [29]. 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.

5.7. 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 [11], 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 previous work [30] 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 5.6. 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.

6. Related Work

The use of model-based development in safety-critical systems to support the development of the system safety case has been the focus of much research. Chen et al. present an approach [31] to integration of model-based engineering with safety analysis to ease the development of safety cases. To overcome the difficulty of information management for advanced and complex systems, the authors present how the architecture description language

EAST-ADL2 can be used to support the development of safety-critical systems. Moreover, to maximise the traceability between the design data, the authors propose a safety case meta-model to connect the GSN classes with EAST-ADL2 entities. Just as in this work, we acknowledge the need for increased support to information management within the development of safety-critical systems. In contrast, in our work, we associate the system domain with the safety case domain through the safety contracts, which support multiple viewpoints of the system. Moreover, we align the information management with the standardised GSN and SACM meta-models to establish the connections between evidence, system, and the safety case arguments.

Wu presents a framework [32] to handle safety concerns and construct safety arguments within a system architectural design process. He presents a set of argument patterns and a method for producing architectural safety arguments. The proposed framework advocates the use of anti-goals and negative scenarios alongside the goals and positive scenarios that are generally used within safety cases. In contrast, we build upon already established argumentation notation and argumentation patterns. Although safety contracts capture scenarios that can be interpreted both as positive and negative, we show only the satisfied contracts, i.e., the positive scenarios, in the generated argument-fragments. The negative scenarios are identified through the safety contracts that are not satisfied and they are handled outside of the safety argument.

Basir et al. present an approach [33] for deriving a safety argument from the actual source code. The authors focus on constructing an argument for how the actual code complies with specific safety requirements based on the V&V artefacts. The argument skeleton is generated from a formal analysis of automatically generated code and integrates different information from heterogeneous sources into a single safety case. The skeleton argument is extended by separately specified additional information enriching the argument with explanatory elements such as contexts, assumptions, justifications etc. In contrast, in our work, we generate argument-fragments from contracts, which essentially describe the actual code on a higher level of abstraction. Moreover, we use the contracts, and their supporting elements captured by SEooCMM, to connect all the heterogeneous sources, which we then exploit for generation of safety case argument-fragments. Although FLAR2SAF facilitates generation of parts of the safety case argument, the contracts platform can be used to provide support for generating other argument-fragments. A complete safety case could be compiled by composing the different argument-fragments. Instead of waiting for the actual code to start generating argument-fragments, we utilise the information about the safety-critical software that is generally available before its development. In this way we support the idea of the safety case as a living document that can be built and reviewed throughout the lifecycle of a system.

Denney et al. focus on automating the assembly of safety cases based on the application of formal reasoning to software [34]. 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 authors use the AutCert tool for formal verification, with the provided specification represented by formalised software requirements. Moreover, the tool includes a meta-model aligned to the standardised SACM and GSN meta-models. Although CHESS toolset does not provide as extensive support for safety case modelling as AutoCert, its integrated support for compositional failure logic analysis gives it an edge when reusing safety artefacts of safety-relevant components. Moreover, we use strong and weak safety contracts as the middle layer between the code and the safety artefacts to provide better support for reuse of safety artefacts of safety components developed out-of-context.

Prokhorova et al. present an approach [35] for deriving safety case arguments that relies on the Event-B formal framework. The authors propose a methodology for formalising the system safety requirements in Event-B and deriving a corresponding safety case argument from the Event-B specification. The authors classify safety requirements by the way they can be represented in Event-B and propose a set of classification-based argument patterns to be used for generating specific arguments for each of the requirements classes. In contrast, we focus on generating argument-fragments for absence of different failure modes, which can be used to increase confidence in satisfaction of different types of safety requirements. By capturing the relationships between requirements, architectural elements, and evidence via safety contracts in SEooCMM, we facilitate capturing of additional information besides the formalised requirements. The increased information management via SEooCMM enables generation of context-specific argument-fragments for reusable components based on the existing argumentation patterns.

7. Conclusion and Future Work

Reuse within safety-critical systems is not complete without reuse of safety artefacts such as argument-fragments and supporting evidence, since they are the key aspects of safety-critical systems development that require significant efforts. In this work we have presented a method called FLAR2SAF for generating reusable argument-fragments. The basis for the argument generation is in the underlying meta-model SEooCMM, which allows for modelling of the out-of-context components together with safety-relevant information captured within safety contracts and supported by the accompanying evidence artefacts. FLAR2SAF first derives safety contracts from failure logic analysis results, and then uses these contracts supported by evidence to generate reusable pattern-based argument-fragments. The lessons learned from applying FLAR2SAF to a real-world case confirm that safety contracts can be derived from failure logic analysis and used to achieve reuse of failure logic analysis related safety artefacts. Moreover, safety contracts translated from FPTC can assist in safety-relevant component selection as they clearly indicate which failures are mitigated under which conditions. Finally, we have shown that the safety contracts translated from FPTC analysis and related to evidence, can be used for generation of context-specific argument-fragments for assuring absence of a particular hazardous software failure mode.

As our future work, in the context of the ECSEL AMASS project, we plan to extend the CHESS toolset to include our methods for derivation of contracts and generation of argument-fragments. Since the FPTC analysis can be computationally demanding, we are working on porting the FPTC analysis and the argument-fragment generation to a cloud setting [36]. Moreover, we plan to explore how different types of safety analyses can be used to derive and support contracts, hence how different types of evidence could be easily reused. Another interesting future direction would be to explore how this approach can help with change management and reuse of safety artefacts in case of changes in the system. More specifically, we are planning to investigate how the strong and weak safety contracts can be used for the safety case maintenance, especially for the safety cases that require frequent updates such as those for autonomous vehicles. While the current approach mainly focuses on the product aspects of safety cases, we plan to extend the SEooCMM and FLAR2SAF to support the process part of the safety cases [37].

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

[1] J. Varnell-Sarjeant, A. A. Andrews, A. Stefik, Comparing Reuse Strategies: An Empirical Evaluation of Developer Views, in: 8th International Workshop on Quality Oriented Reuse of Software, IEEE, 498–503, 2014.

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

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

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

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

[6] B. Gallina, M. A. Javed, F. U. Muram, S. Punnekkat, Model-driven Dependability Analysis Method for Component-based Architectures, in: 38th Euromicro Conference on Software Engineering and Advanced Applications, IEEE, 233–240, 2012.

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

[8] I. Sljivo, B. Gallina, J. Carlson, H. Hansson, Strong and Weak Contract Formalism for Third-Party Component Reuse, in: 3rd International Workshop on Software Certification, International Symposium on Software Reliability Engineering Workshops (ISSREW), IEEE, 359–364, 2013.

[9] I. Sljivo, B. Gallina, J. Carlson, H. Hansson, Generation of Safety Case Argument-Fragments from Safety Contracts, in: 33rd International Conference on Computer Safety, Reliability, and Security, vol. 8666 of *Lecture Notes in Computer Science*, Springer, 170–185, 2014.

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

[11] M. Wallace, Modular Architectural Representation and Analysis of Fault Propagation and Transformation, Electronic Notes in Theoretical Computer Science 141 (3) (2005) 53 – 71.

[12] B. Gallina, 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, 493–500, 2011.

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

[14] R. Dardar, B. Gallina, A. Johnsen, K. Lundqvist, M. Nyberg, Industrial Experiences of Building a Safety Case in Compliance with ISO 26262, in: 2nd International Workshop on Software Certification, International Symposium on Software Reliability Engineering Workshops (ISSREW), IEEE, 349–354, 2012.

[15] B. Gallina, A. Gallucci, K. Lundqvist, 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), 2013.

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

[17] M. Bender, T. Maibaum, M. Lawford, A. Wassyng, Positioning Verification in the Context of Software/System Certification, Electronic Communications of the EASST 46.

[18] O. M. G. (OMG), SACM: Structured Assurance Case Metamodel, Tech. Rep., Version 1.0, OMG. http://www.omg.org/spec/SACM, 2013.

[19] GSN Working Group, GSN Metamodel Draft, `http://www.goalstructuringnotation.info/archives/291`, 2014.

[20] Bishop, P. and Bloomfield, R., A Methodology for Safety Case Development, in: Redmill, F. and Anderson, T. (Ed.), Industrial Perspectives of Safety-critical Systems: 6th Safety-critical Systems Symposium, Springer, 194–203, 1998.

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

[22] B. Gallina, S. Kashiyarandi, K. Zugsbrati, A. Geven, Enabling Cross-domain Reuse of Tool Qualification Certification Artefacts, in: International Workshop on Development, Verification and Validation of Critical Systems, vol. 8696 of *Lecture Notes in Computer Science*, Springer, 255–266, 2014.

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

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

[25] R. Stake, Case Studies, in: N. K. Denzin, Y. S. Lincoln (Eds.), Handbook of Qualitative Research, chap. 14, Sage Publications, Oxford, 236–247, 1994.

[26] I. Sljivo, B. Gallina, J. Carlson, H. Hansson, Using Safety Contracts to Guide the Integration of Reusable Safety Elements within ISO 26262, in: 21st IEEE Pacific Rim International Symposium on Dependable Computing, IEEE, 129–138, 2015.

[27] D. Parker, M. Walker, Y. Papadopoulos, Model-Based Functional Safety Analysis and Architecture Optimisation, Embedded Computing Systems: Applications, Optimization, and Advanced Design: Applications, Optimization, and Advanced Design (2013) 79–92.

[28] I. Habli, T. Kelly, Balancing the Formal and Informal in Safety Sase Arguments, in: VeriSure: Verification and Assurance Workshop, colocated with Computer-Aided Verification (CAV), 2014.

[29] J. Rushby, Logic and Epistemology in Safety Cases, in: 32nd International Conference on Computer Safety, Reliability, and Security, vol. 8153 of *Lecture Notes in Computer Science*, Springer, 1–7, 2013.

[30] I. Sljivo, B. Gallina, J. Carlson, H. Hansson, S. Puri, A Method to Generate Reusable Safety Case Fragments from Compositional Safety Analysis, in: 14th International Conference on Software Reuse, Springer, 253–268, 2015.

[31] D. Chen, R. Johansson, H. Lönn, Y. Papadopoulos, A. Sandberg, F. Törner, M. Törngren, Modelling Support for Design of Safety-critical Automotive Embedded Systems, in: 27th International Conference on Computer Safety, Reliability, and Security, vol. 5219 of *Lecture Notes in Computer Science*, Springer, 72–85, 2008.

[32] W. Wu, Architectural Reasoning for Safety — Critical Software Applications, Ph.D. thesis, University of York, York, UK, 2007.

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

[34] E. Denney, G. J. Pai, Automating the Assembly of Aviation Safety Cases, IEEE Transactions on Reliability 63 (4) (2014) 830–849.

[35] Y. Prokhorova, L. Laibinis, E. Troubitsyna, Facilitating Construction of Safety Cases from Formal Models in Event-B, Information & Software Technology 60 (2015) 51–76.

[36] S. Alajrami, B. Gallina, I. Sljivo, A. Romanovsky, P. Isberg, Towards Cloud-Based Enactment of Safety-Related Processes, in: 35th International Conference on Computer Safety, Reliability and Security, Springer, 2016.

[37] B. Gallina, A Model-driven Safety Certification Method for Process Compliance, in: 2nd International Workshop on Assurance Cases for Software-intensive Systems, IEEE, 204–209, 2014.