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

Irfan Sljivo[1], Barbara Gallina[1], Jan Carlson[1], Hans Hansson[1] and Stefano Puri[2]

[1] Mälardalen Real-Time Research Centre, Mälardalen University, Västerås, Sweden
{irfan.sljivo, barbara.gallina, jan.carlson, hans.hansson}@mdh.se
[2] Intecs, SpA, Pisa, Italy
stefano.puri@intecs.it

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

Keywords: Component- and contract-based architectures, Compositional safety analysis and argumentation, Safety argumentation reuse.

1 Introduction

A recent study within the US Aerospace Industry shows that reuse is more present when developing embedded systems than non-embedded systems [16]. The study reports that code is reused most of the time, followed by requirements and architectures in significantly smaller scale than code. Aerospace industry, as most other safety-critical industries, needs to follow a domain specific safety standard that requires additional artefacts to be provided alongside the code to show that the code is acceptably safe to operate in a given context. The costs of producing the verification artefacts are estimated at more than 100 USD per code line, while for highly critical applications the costs can reach up to 1000 USD [4]. In most cases, as part of the certification efforts an additional time-consuming and expensive task of providing a safety case is required. A safety case

is documented in form of an explained and well-founded structured argument to clearly communicate that the system is acceptably safe to operate in a given context [13].

Most safety standards are starting to acknowledge the need for reuse, hence the latest versions of both aerospace (DO178-C) and automotive (ISO 26262) industry standards explicitly support techniques for reuse, e.g., the notion of Safety Element out of Context (SEooC) within automotive [12] and Reusable Software Components (RSC) within aerospace industry [1]. This allows for easier integration of reusable components, such as Commercial of the shelf (COTS), but it also means that some safety artefacts of the reused components should be reused as well if we are to fully benefit from the reuse and safely integrate the reused component into the new system. The difficulty that hinders reuse is that safety is a system property. This means that hazard analysis and risk assessment used to analyse what can go wrong at system level, as required by the standards, can only be performed in a context of the specific system. To overcome this difficulty compositional approaches are needed. CHESS-FLA [7] is a plugin within the CHESS toolset [6] that supports execution of Failure Logic Analysis (FLA) such as Fault Propagation and Transformation Calculus (FPTC). FPTC allows us to calculate system level behaviour given the behaviour of the individual components established in isolation. Such compositional failure analyses enable reuse of safety artefacts within safety-critical systems.

Component-based Development (CBD) is the most commonly used approach to achieve reuse within embedded systems of the aerospace industry [16]. While CBD is successfully used to support reuse of software components, it lacks means to support reuse of additional artefacts, alongside the software components, in form of argument-fragments and supporting evidence. As a part of an overall system safety argument, argument-fragments for software components present safety reasoning used to develop a particular component and its safety-relevant behaviour, e.g., failure behaviour.

In our previous work we developed the notion of safety contracts related to software components to promote reuse of the components together with their certification data and we have proposed a (semi)automatic method to generate argument-fragments for the software components from their associated safety contracts [14]. In this work we propose a method called FLAR2SAF that uses failure logic analysis results (FLAR) to generate safety case argument-fragments (SAF). More specifically, we derive safety contracts for a component from FLAR. Then, we adapt our method for generation of argument-fragments to provide better support for reuse of the argument-fragments and the evidence they contain.

In particular, the input/output behaviour of a component developed out-of-context can be specified by FPTC rules. For example, in case of omission failure on the input I1 of the component, the component can have a safety mechanism to still provide the output O1 but with additional delay. In that case FPTC rule describing such behaviour can be specified as: $I1.omission \rightarrow O1.late$. We can use these behaviours obtained by FPTC analysis to derive safety contracts that can be further supported by evidence and used to form clear

and comprehensive argument-fragments. For example, if the late failure on the output of the component can cause a hazardous event, then the corresponding argument-fragment should argue that the late failure is sufficiently handled in the context of the particular system and attach supporting evidence for that claim. For generating argument-fragments associated to the failure behaviour of the components we use an established argument pattern [18].

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

The rest of the paper is organised as follows: In Section 2 we provide background information. In Section 3 we present the rationale behind our approach and methods to derive safety contracts from FPTC analysis and generate corresponding argument-fragments. In Section 4 we illustrate our approach by applying it to a wheel-braking system. We present the related work in Section 5, and conclusions and future work in Section 6.

2 Background

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

2.1 COTS-based safety-critical architectures

In the context of safety critical systems, COTS-driven development is becoming more and more appealing. The typical V model that constitutes the reference model for various safety standards is being combined with the typical component-based development. As Fig.1 depicts, the top-down and bottom-up approach meet in the gray zone. Initially a top-down approach is carried out. The typical safety process starts with hazards identification which is conducted by analysing (brainstorming on) failure propagation, based on an initial description of the system and its possible functional architecture. If a failure at system level may lead to intolerable hazards, safety requirements are formulated, decomposed onto the architectural components, and mitigation means have to be designed. Safety requirements are assigned with Safety Integrity Levels (SILs) as a measure of quantifying risk reduction. Iteratively and incrementally the system architecture is changed until a satisfying result is achieved (i.e. no intolerable behaviour at system level). More specifically, once the safety requirements are decomposed onto components (hardware/software), COTS (developed via a bottom-up approach) can be selected to meet those requirements. If the selected components do not fully meet the requirements, some adaptations can be introduced.

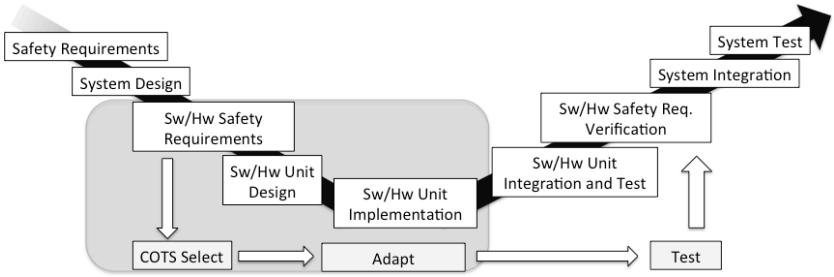

Fig. 1. Safety-critical system development/COTS-driven development

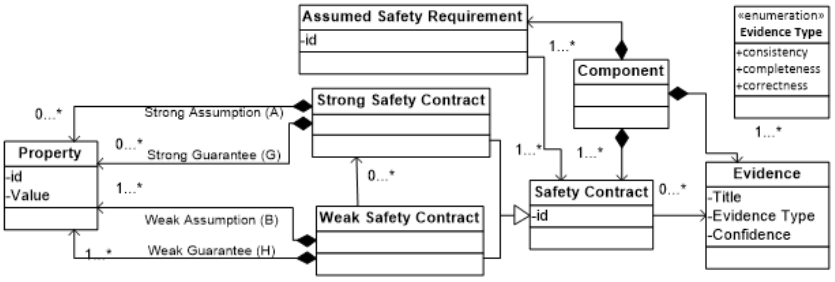

Fig. 2. Component and safety contract meta-model [14]

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 [15]. A traditional component contract $C = \langle A, G\rangle$ is composed of assumptions (A) on the environment of the component and guarantees (G) that are offered by the component if the assumptions are met. The strong contract assumptions (A) are required to be satisfied in all contexts in which the component is used, hence the corresponding strong guarantees (G) are offered in all contexts in which the component can be used. For example, a strong assumption could be minimum amount of memory a component requires to operate. The weak contract guarantees (H) are offered only in those contexts where besides the strong assumptions, the corresponding weak assumptions (B) are satisfied as well. This makes the weak contracts context specific, e.g., a timing behaviour of a component on a specific platform is captured by a weak contract.

We denote a contract capturing safety-relevant behaviour as a safety contract. In [14] we introduced a component meta-model (Fig. 2) that connects safety contracts with supporting evidence, which provides a base for evidence artefact reuse together with the contracts. The component meta-model specifies a component in an out-of-context setting composed of safety-contracts, evidence and the assumed safety requirements. Each safety requirement is satisfied by at least one safety contract, and each contract can be supported by one or more ev-

idence. For example, if we assume that late output failure of the component can be hazardous, then we define an assumed safety requirement that specifies that late failure should be appropriately handled. This requirement is addressed by a contract that captures in its assumptions the identified properties that need to hold for the component to guarantee that the late failure is appropriately handled. The evidence that supports the contract includes contract consistency report and analyses results used to derive the contract.

2.2 CHESS-FLA within the CHESS toolset

CHESS-FLA [7] is a plugin within the CHESS toolset [6] that includes two FLA techniques: (1) FPTC [17] - a compositional technique to qualitatively assess the dependability of component-based systems, and (2) FI^{4}FA [9] - FPTC extension that allows for analysis of mitigation behaviour. In this paper we limit our attention to FPTC that allows users to calculate the behaviour at system-level, based on the specification of the behaviour of individual components. In the CHESS toolset components can be modelled as component types or component implementations. Component types are more abstract and can be realised by system-specific component implementations. Component implementations inherit all behaviours of the corresponding component type.

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

The syntax supported in CHESS-FLA to specify the FPTC rules is shown in Fig. 3. An example of a compliant expression that demonstrates the transformational behaviour of a component is "$R1.late \rightarrow P1.valueCoarse$", which should be read as follows: if the component receives on its port R1 a late failure, it generates on its output port P1 a coarse (i.e. clearly detectable) value failure (a failure that manifests itself as a failure mode by exceeding the allowed range).

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

Fig. 3. FPTC syntax supported in CHESS-FLA

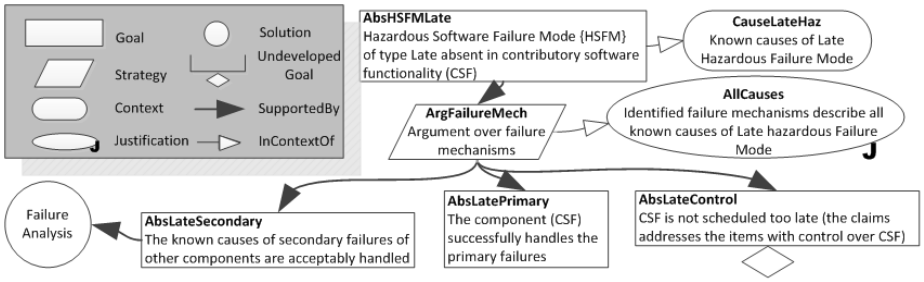

Fig. 4. Hazardous Software Failure Mode absence pattern for type late failure

2.3 Safety cases and safety case modelling

A Safety case in form of an explained (argued about) and well-founded (evidence-based) structured argument is often required to show that the system is acceptably safe to operate in a given context [13]. Goal Structuring Notation (GSN) is a graphical argumentation notation for documenting the safety case [10]. GSN can be used to represent the individual elements of any safety argument and the relationships between these elements. The argument usually starts with a top-level claim/goal stating absence of a failure, as in Fig. 4 the argument starts with a goal that has *AbsHSFMLate* identifier. The goals can be further decomposed to sub-goals with *supportedBy* relations denoting inference between goals or connecting supporting evidence with a goal. The decomposition can be described using strategy elements e.g., *ArgFailureMech* in Fig. 4. To define the scope and context of a goal or provide its rationale, elements such as context and justification are attached to a goal with *inContextOf* relations. For example, context *CauseLateHaz* is used to clarify the *AbsHSFMLate* goal by providing the list of known causes of the late failure mode. The undeveloped element symbol indicates elements that need further development. For more details on GSN see [10].

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 [10]. In this work we use the argument pattern for Handling of Software Failure Modes (HSFM) [18], a portion of which is shown in Fig. 4, to structure the generated argument-fragments related to late timing failure modes. To build an argument, HSFM pattern requires information about known causes of the failure mode and failure mechanisms that address those causes. Moreover, the failure mechanisms can be classified into three categories: (1) Primary failures within Contributory Software Functionality (CSF) that can cause the failure; (2) Secondary failures relating to other components within the system on which CSF is dependent; and (3) Failures caused by items controlling CSF e.g., in case of late hazardous failure mode the controlling item is the scheduling policy.

3 FLAR2SAF

In this section we present FLAR2SAF, a method to generate reusable safety case argument-fragments. We first provide the rationale of the approach in Section 3.1. We provide a method to translate FPTC rules into safety contracts in Section 3.2, and we adapt and extend the method for semi-automatic generation of argument-fragments from safety contracts in Section 3.3.

3.1 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 evidence needs to be provided with a reusable component to increase confidence in the component itself, similarly in some cases the trustworthiness of the evidence should be backed up as well [11]. To reuse evidence-related artefacts together with the argument fragments, additional information about the rationale linking the artefacts and the safety contracts they support should be provided. Furthermore, the issue of trustworthiness of such evidence needs to be addressed. For example, we might need to describe the competence of the engineers that performed a particular analysis or even qualification of the analysis tool.

To capture the additional information related to evidence we enrich the component meta-model presented in Section 2.1. We enrich the connection between a contract and evidence by adding optional descriptive attribute capturing the rationale for how the particular evidence, or set of evidence, supports the goal. This information is used to provide additional clarification on the connection between the evidence and the claims made by the contract. Clarification of confidence in the evidence itself can be made in two different ways: either by directly including or referencing supporting information in the context of the evidence (e.g., competence of person performing the failure analysis can be found in document x); or to point to an already developed goal, called an *away goal* [10], presenting the supporting information (we could have a repository of generic argument-fragments related to staff competence and tool-qualification [8]). In the presented component meta-model we append attributes to the evidence to capture supporting information related to the evidence, including a set of references to the supporting away goals.

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

- Model the component architecture in CHESS-FLA;
- Specify failure behaviour of a component in isolation using FPTC rules;
- Translate the FPTC rules into corresponding safety contracts and attach FPTC analysis results as initial evidence;
- Support the contracts with additional V&V evidence and enrich the contract assumptions accordingly;
- Upon component selection, depicted in Fig. 1 in Section 2.1:

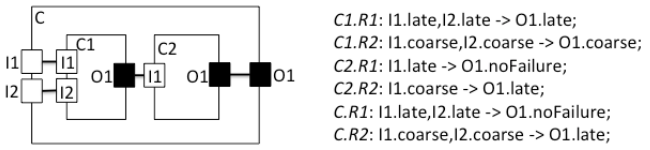

Fig. 5. Composite component example with FPTC rules

- Perform FPTC analysis and calculate system-level failure behaviour;
- Translate the results of FPTC analysis to system-level safety contracts;
- Support and enrich the contracts with additional V&V evidence;

– Use the approach to semi-automatically generate an argument-fragment based on the argument pattern presented in Section 2.3.

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

3.2 Contractual interpretation of FPTC rules

In this section we focus on the step of translating the FPTC rules to safety contracts. We use the simple example in Fig. 5 to explain the translation process and provide a set of steps that can be used to perform the translation

In Fig. 5 we have FPTC rules specified for a composite component C and its subcomponents C1 and C2. When both inputs I1 and I2 exhibit late or coarse failure, component C1 acts as a propagator and outputs late/coarse failure on O1 output. Component C2 acts as a sink in case of a late failure and transforms it to no failure (e.g., a watchdog timer expires and triggers a satisfactory response), while it transforms coarse to late failure (e.g., due to additional filtering).

Safety contracts for these components can be made based on the FPTC rules. When translating the rules into contracts we consider two types of rules with respect to each failure mode: rules that describe when a failure happens (e.g., C1.R1) and rules that describe behaviours that mitigate a failure (e.g., C2.R1). We translate the first type of rules by guaranteeing with the contract that the failure described by the rule will not happen, under assumptions that the behaviour that causes the failure does not happen. The contract $\langle B, H \rangle_{C1}$ for component C1, shown in Table 1, guarantees that O1 will not be late if both inputs I1 and I2 never fail at the same time with late failure. This type of contracts is specified as weak since, unlike for strong contracts, their satisfaction in every context should not be mandatory. For example, in some contexts late timing failure is not hazardous, hence it is not required to be ensured.

We translate the second type of rules differently as they do not identify causes of failures, but they specify behaviours that help mitigate failures in certain

Table 1. Contracts for components C1 and C

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

cases. Since these contracts specify safety behaviour of components that should be satisfied in every context, without imposing assumptions on the environment, we denote these contracts as strong contracts. The corresponding contracts state in which cases the component guarantees that it will not exhibit any failures. We do this by guaranteeing the rule that describes this behaviour, as shown in Table 1 for the $\langle A, G \rangle_{C-1}$ contract for component C.

As shown on an example of translating FPTC rules from the example in Fig. 5 to contracts in Table 1, the translation can be performed in the following way for each failure:

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

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

3.3 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 previously established argument pattern HSFM for presenting absence of late 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 be additionally added to the safety contracts, e.g., scheduler policy constraints. We construct the argument-fragment by using the reasoning from the HSFM pattern. The top-most goal claiming absence of the failure mode is decomposed into three sub-goals focusing on primary, secondary and controlling failures as described in Section 2.3. We adapt the contract-satisfaction fragment from [14] to further develop the sub-goals.

We use the safety contracts to generate the supporting sub-arguments for the primary and secondary failures and leave the goal related to controlling failures undeveloped. Supporting sub-arguments for both primary and secondary failures are generated to argue that the corresponding safety contracts are satisfied with sufficient confidence. The sufficient confidence is determined based on the specific SIL of the requirements allocated on the component and may require additional evidence in case of higher SILs. We argue the satisfaction of contracts as in [14] where we make a claim that the contract is satisfied with sufficient confidence, i.e., that the guarantee of the contract is offered. We further decompose the claim into two supporting goals: (1) an argument providing the supporting evidence for confidence in the claim in terms of completeness of the contract, and (2) an argument showing that the assumptions stated in the contract are met by the contracts of other components. We further focus on the first sub-goal related to evidence and adapt the rules related to generating the evidence sub-argument to include additionally specified information about the evidence artefacts.

For every evidence attached to a safety contract we create a sub-goal to support confidence in the corresponding safety contract. At this point we can use the additional information about the rationale connecting evidence and the safety contract and present it in form of a context statement to clarify how this particular evidence contributes to increasing confidence in the corresponding safety contract. The evidence can be further backed up by the related 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 [14] to allow for argument-fragment generation in the specific form of the selected pattern. The approach is adapted to generate an argument-fragment that clearly separates and argues over primary, secondary and controlling failures as described above, and to include additional information related to the evidence.

While the benefits of reusing evidence are great, a big risk can be falsely reusing evidence which may result in false confidence and potentially unsafe sys-

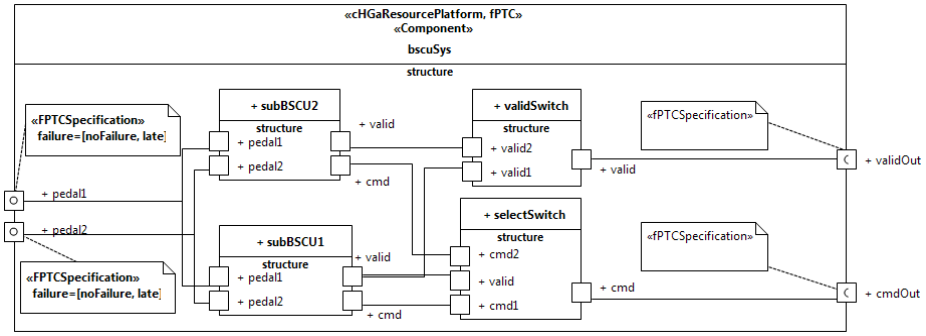

Fig. 6. BSCU model in CHESS

tem. 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.

4 Application Example

In this section we demonstrate FLAR2SAF by applying it to a Wheel-Braking System (WBS). We first briefly introduce the WBS in Section 4.1. In Section 4.2 we apply CHESS-FLA/FPTC analysis on WBS. We use the translation steps from Section 3.2 to translate the contracts from the FPTC analysis results in Section 4.3. We present the generated argument-fragment in Section 4.4.

4.1 Wheel Braking System (WBS)

In this section we recall WBS, which was originally presented in ARP4761 [2]. We use a simplified version of WBS to illustrate the use of FLAR2SAF.

WBS is a part of an airplane braking system. It takes two input brake pedal signals that are used by the Brake System Control Unit (BSCU) to calculate the braking force. The software architecture of BSCU modelled in CHESS is shown in Fig. 6. Based on the preliminary safety analysis performed on the system, the BSCU is designed with two redundant dual channel systems to meet the availability and integrity requirements. Each of the two subBSCU systems, namely subBSCU1 and subBSCU2, provide a calculated command value and a valid signal that indicates the validity of the corresponding command value. The selectSwitch forwards by default the command value from subBSCU1 if the corresponding valid signal is true, otherwise the command value from subBSCU2 is forwarded. The validSwitch component returns true if any of the signals is true,

Table 2. A subset of FPTC rules for BSCU subcomponents

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

otherwise it returns false indicating that an alternate braking mode should be used, as the braking command calculated by BSCU cannot be trusted.

4.2 FPTC analysis

To perform the FPTC analysis we first model the system architecture in the CHESS-toolset (Fig. 6) and then define FPTC rules for the modelled components. The architecture and the corresponding failure behaviour of the components are defined based on the system description in Section 4.1.

The specified FPTC rules are shown in Table 2. As mentioned in Section 2.2, the FPTC rules specified for components are inherited by all the instances, hence the FPTC rules for the two subBSCU component implementations are the same as they are instances of the same component. The validSwitch component requires at least one valid signal present in order to forward the correct response, i.e., at least to signal that there is a problem within BSCU. Similarly, the selectSwitch component output depends both on valid and cmd signals.

As shown in Fig. 6 in the FPTC specifications on the input ports, we run the analysis for noFailure and late failure behaviours on the inputs. The FPTC analysis then computes the possible failures on the output ports of BSCU based on the FPTC rules for the BSCU subcomponents. The results show that the validOut port can either not fail or propagate late failures, while the cmdOut port in addition to noFailure and late failure can exhibit omission failure as well.

Table 3. The results of the FPTC analysis for bscuSys component

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

Table 4. The translated BSCU contracts and associated evidence information

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

4.3 The translated contracts

The results of the FPTC analysis can be interpreted in the form of FPTC rules for the system component *bscuSys*. The resulting FPTC rule "pedal1.late, pedal2.late → validOut.late, cmdOut.late" for *bscuSys* can be translated to the contract $\langle B, H \rangle_{BSCU-1}$ shown in Table 4. The contract specifies that the outputs of BSCU will not be late if both input pedals are not late. The contract is supported by the FPTC analysis report from which the contract is derived.

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

4.4 The resulting argument-fragment

A part of the resulting argument-fragment is shown in Fig. 7. In this argument snippet we focus only on the identified causes of primary failures (*AbsLatePrimary* goal), while the other goals shown in Fig. 4 remain undeveloped. We identified the BSCU-2 contract shown in Table 4 as the one related to primary failures as it describes behaviour of the component that mitigates a possible failure. By applying the rules to generate the contract satisfaction argument (goal *BSCU-2_sat*), we divide the argument to argue over the satisfaction of the supporting contracts (*BSCU-2_supp_sat*) and supporting evidence in contract completeness

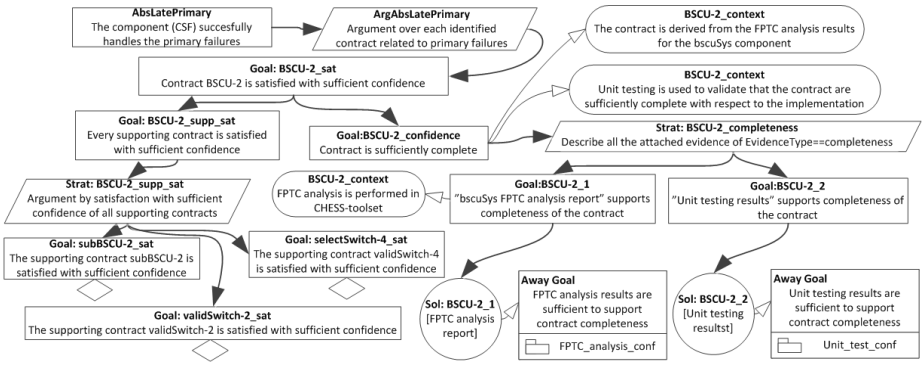

Fig. 7. Argument-fragment based on the HSFM pattern

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

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

5 Related Work

The use of model-based development in safety-critical systems to support the development of the system safety case has been the focus of much research during the past years. Integration of model-based engineering with safety analysis to ease the development of safety cases is presented in [5]. The work presents how the architecture description language EAST-ADL2 can be used to support the development of safety-critical systems. Similarly, an approach to handling safety concerns and constructing safety arguments within a system architectural design process is presented in [19]. The work presents a set of argument patterns and a supporting method for producing architectural safety arguments. The focus of these works is usually on extending the modelling approaches to support the safety case development process and provide guidelines on how to produce the corresponding safety arguments. Unlike in these approaches, in our work we

provide a method for generating safety-arguments from the safety contracts that are based on and supported by the safety analysis performed on the system.

Deriving a safety argument from the actual source code is presented in [3]. The work focuses on constructing an argument for how the actual code complies with specific safety requirements based on the V&V artefacts. The argument skeleton is generated from a formal analysis of automatically generated code and integrates different information from heterogeneous sources into a single safety case. The skeleton argument is extended by separately specified additional information enriching the argument with explanatory elements such as contexts, assumptions, justifications etc. In contrast, in this work we generate an argument-fragment from safety contracts obtained from and supported by FPTC analysis. We utilise the contracts to specify the additional information regarding the context and additional assumptions and generate an argument-fragment for a specific failure mode covered by the FPTC analysis.

6 Conclusion and Future Work

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

As our future work we are planning an evaluation of FLAR2SAF on an industrial case study. Moreover, we plan to extend the CHESS toolset to include our methods for derivation of contracts and generation of argument-fragments. We plan to explore how different types of safety analyses can be used to derive and support contracts, hence how different types of evidence could be easily reused. Another interesting future direction would be to explore how this approach can help us with change management and reuse of safety artefacts in case of changes in the system.

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

References

1. AC 20-148: Reusable Software Components. FAA (2004)
2. ARP4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment. Society of Automotive Engineers (1996)
3. Basir, N., Denney, E., Fischer, B.: Building heterogeneous safety cases for automatically generated code. In: Infotech@ Aerospace Conference. AIAA (2011)
4. Bloomfield, R., Cazin, J., Craigen, D., Juristo, N., Kesseler, E., et al.: Validation, Verification and Certification of Embedded Systems. Tech. rep., NATO (2005)
5. Chen, D., Johansson, R., Lönn, H., Papadopoulos, Y., Sandberg, A., Törner, F., Törngren, M.: Modelling support for design of safety-critical automotive embedded systems. In: Harrison, M., Sujan, M.A. (eds.) 27th International Conference on Computer Safety, Reliability, and Security. LNCS, vol. 5219, pp. 72–85. Springer, Heidelberg (2008)
6. CHESS-toolset, `http://www.chess-project.org/page/download`
7. Gallina, B., Javed, M.A., Muram, F.U., Punnekkat, S.: Model-driven Dependability Analysis Method for Component-based Architectures. In: Euromicro-SEAA Conference. IEEE Computer Society (2012)
8. Gallina, B., Kashiyarandi, S., Zugsbrati, K., Geven, A.: Enabling cross-domain reuse of tool qualification certification artefacts. In: Bondavalli, A., Ceccarelli, A., Ortmeier, F. (eds.) International Workshop on Development, Verification and Validation of Critical Systems. LNCS, vol. 8696, pp. 255–266. Springer, Heidelberg (2014)
9. Gallina, B., Punnekkat, S.: FI^{4}FA: A Formalism for Incompletion, Inconsistency, Interference and Impermanence Failures Analysis. In: International workshop on Distributed Architecture modeling for Novel Component based Embedded systems. IEEE Computer Society (2011)
10. GSN Community Standard Version 1. Origin Consulting (York) Limited (2011)
11. Hawkins, R., Habli, I., Kelly, T., McDermid, J.: Assurance cases and prescriptive software safety certification: A comparative study. Safety science 59, 55–71 (2013)
12. ISO 26262:2011: Road vehicles — Functional safety. International Organization for Standardization (2011)
13. Kelly, T.P.: Arguing Safety — A Systematic Approach to Managing Safety Cases. Ph.D. thesis, University of York, York, UK (1998)
14. Sljivo, I., Gallina, B., Carlson, J., Hansson: Generation of Safety Case Argument-Fragments from Safety Contracts. In: Bondavalli, A., Di Giandomenico, F. (eds.) 33rd International Conference on Computer Safety, Reliability, and Security. LNCS, vol. 8666, pp. 170–185. Springer, Heidelberg (2014)
15. Sljivo, I., Gallina, B., Carlson, J., Hansson, H.: Strong and weak contract formalism for third-party component reuse. In: International Workshop on Software Certification. IEEE Computer Society (2013)
16. Varnell-Sarjeant, J., Andrews, A.A., Stefik, A.: Comparing Reuse Strategies: An Empirical Evaluation of Developer Views. In: International Workshop on Quality Oriented Reuse of Software. IEEE Computer Society (2014)
17. Wallace, M.: Modular architectural representation and analysis of fault propagation and transformation. In: International Workshop on Formal Foundations of Embedded Software and Component-based Software Architectures. Elsevier (2005)
18. Weaver, R., McDermid, J., Kelly, T.: Absence of Late Hazardous Failure Mode, `http://www.goalstructuringnotation.info/archives/218`
19. Wu, W.: Architectural Reasoning for Safety — Critical Software Applications. Ph.D. thesis, University of York, York, UK (2007)