Tool-Supported Safety-Relevant Component Reuse: From Specification to Argumentation

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

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

Abstract. Contracts are envisaged to support compositional verification of a system as well as reuse and independent development of their implementations. But reuse of safety-relevant components in safety-critical systems needs to cover more than just the implementations. As many safety-relevant artefacts related to the component as possible should be reused together with the implementation to assist the integrator in assuring that the system they are developing is acceptably safe. Furthermore, the reused assurance information related to the contracts should be structured clearly to communicate the confidence in the component. In this work we present a tool-supported methodology for contract-driven assurance and reuse. We define the variability on the contract level in the scope of a trace-based approach to contract-based design. With awareness of the hierarchical nature of systems subject to compositional verification, we propose assurance patterns for arguing confidence in satisfaction of requirements and contracts. We present an implementation extending the AMASS platform to support automated instantiation of the proposed patterns, and evaluate its adequacy for assurance and reuse in a real-world case study.

1 Introduction

Software-intensive systems are rarely developed from scratch. Instead, components developed previously are reused for building new systems [1]. The same trend is visible in safety-critical systems, which usually need to be assured that they are acceptably safe to be deployed. The assurance entails gathering a body of evidence in form of a safety assurance case to communicate that any unreasonable risk in the system has been mitigated. Due to this, reuse of components in such systems is not complete without the reuse of assurance information associated with the component. While reuse of safety-related components is very much present in safety-critical systems development, the lack of systematic approaches to managing reuse of both components and their accompanying assurance information has shown to be dangerous in the past [2].

To address the issue of reuse in safety-critical systems, some reuse principles have been promoted through the safety standards. For example, the automotive functional safety standard ISO 26262 [3] with its concept of Safety Element out-of-Context (SEooC) for reuse of components together with the related safety assurance information. It promotes principles that should be followed to begin the assurance process on the level of the SEooC, which is being developed independently from the system in which it will be used. The purpose of the early start of the assurance process is to support the integrator of the SEooC in assuring their system according to the standard. Ideally, if all suppliers would provide their components as SEooC, the integrator should have an easier job of assuring that the integrated system is acceptably safe. The core aspect of SEooC development are assumptions on the context in which the SEooC component could be reused, such that their validation upon reuse establishes whether the component and the related assurance information is reusable in the particular context.

To support SEooC development and reuse, we have proposed to use *assumption/guarantee component contracts* in our previous work [4]. A contract is a pair of assertions called assumptions and guarantees, where the component guarantees a certain behaviour, given that the environment in which it is deployed fulfils the assumptions [5]. Such contracts provide a systematic way to capture the context assumptions and relate them with the properties that the SEooC component implements. We have proposed to relate contracts with the assurance information [4] and support contract-driven assurance by automating the generation of assurance argument-fragments on satisfaction of both such contracts and the system requirements that can be validated via those contracts.

Reusable components such as SEooC are often characterised with parameters that are used to tailor the behaviour of the component in the different settings in which the component is reused. To address such need for variability at the contract level, we have made a distinction between strong and weak contracts [6]. On the one hand, the strong contracts are those whose assumptions should be met by every context in which the component is reused, hence its guarantees are always offered by the component. On the other hand, the weak contract assumptions do not need to be satisfied by every context in which the component is reused, but when they are met, only then the component offers the corresponding weak guarantees. This variability on the contract level can be used to identify which assumed safety requirements offered by the SEooC component are relevant in the system in which the SEooC is reused. Hence, the safety case information related to those requirements and contracts can also be identified for reuse. To set the ground for tool support, we have proposed a generic SEooC MetaModel (SEooCMM) that defines relationships between SEooC components, contracts, requirements and assurance assets [4]. The basic elements needed for the tool support are a system modelling tool compliant with the SEooCMM, a contract checking engine, and a safety case modelling tool.

In this paper we present our efforts to provide tool-support for contract-based design that incorporates strong and weak contracts as well as the automated generation of assurance arguments. We turn to the AMASS[1] platform for our imple-

[1] AMASS - Architecture-driven, Multi-concern and Seamless Assurance and Certification of Cyber-Physical Systems, https://amass-ecsel.eu/

mentation as it includes the needed tools for system modelling (CHESS[2]), contract checking (OCRA[3]) and safety assurance case modelling (OpenCert[4]). Two challenges arise when using the AMASS platform for contract-driven reuse and assurance: (1) the contract-based design framework [7] implemented in OCRA does not distinguish between the strong and weak contracts; (2) the connection between the system and assurance modelling domains is not clearly defined. To address the first challenge, we define the strong and weak contracts in the scope of the contract-based framework implemented in OCRA. Moreover, we present how refinement checking can be adapted to support strong and weak contracts through the interaction of CHESS and OCRA. To address the second challenge, we first identify the information needed to perform contract-driven assurance and extend CHESS to allow for its modelling. We structure that information by extending the argument pattern for assurance of contract satisfaction to account for the hierarchical component decomposition defined through the notion of refinement. Then, we develop a transformation from the system model to the assurance model that automatically instantiates the defined argument-fragment for each component in the system. Finally, we validate the tool-supported contract-based assurance and reuse methodology in a real-world case study.

As assurance cases are gaining popularity, there is an increasing number of tools supporting their development with particular focus on automation capabilities. For example, *Safety.Lab* [8] focuses on model-based safety analysis and generates an argument structure from rich models of various safety-relevant artefacts. The Eclipse-based *Resolute* tool [9] facilitates generating assurance arguments from architectural models. The *Evidence Confidence Assessor (EviCA)* [10] is a diagramming tool that supports automated generation of confidence arguments related to manually created arguments. The *AdvoCATE* [11] toolset includes a variety of automated features for assurance case creation and analysis. AdvoCATE automates instantiation of pre-developed argumentation pattern from a hazard and safety requirement analysis. While we also automatically instantiate a pre-developed pattern, we do so from architectural models enriched with assumption/guarantee contracts coupled with safety-relevant artefacts. This allows us to filter the relevant artefacts and provide additional support for reuse and tailoring of context-specific automated argument generation.

The rest of the paper is organised as follows: In Section 2, we present some background information. We present the tool-supported methodology for contract-driven assurance and reuse in Section 3. In Section 4, we present our case study. Finally, we bring conclusions and indicate future work in Section 5.

2 Background

In this section we first present the tools and concepts we build upon, and then we present the system description of the considered case study.

[2] https://www.polarsys.org/chess
[3] https://ocra.fbk.eu/
[4] https://www.polarsys.org/projects/polarsys.opencert

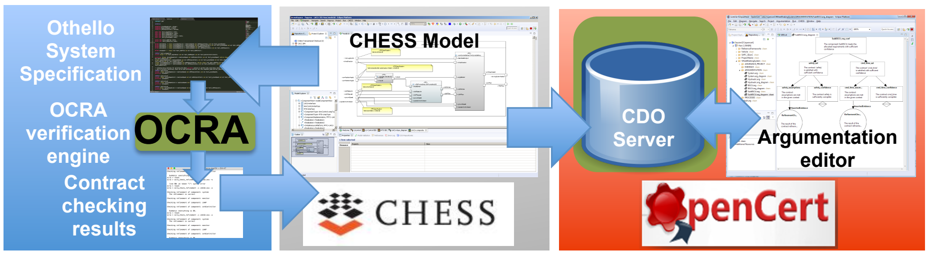


Fig. 1. The overview of the tool information flow

2.1 AMASS Platform

The AMASS platform encompasses different tools, but we focus on the three tools that facilitate system modelling (CHESS), formal verification of assumption guarantee contracts (OCRA), and assurance case modelling (OpenCert). An overview of the three tools is shown in Fig. 1. In the reminder of the section, we present the tools together with their underlying theoretical concepts.

System Modelling: CHESS provides an editor to model all phases of system development: from requirements definition, architecture modelling to software design and its deployment to hardware. In the CHESS toolset, components can be modelled as component types or component instances. Component types can be seen as elements out of context, and component instances as the in-context representation of the corresponding component types. Component instances inherit the attributes of the corresponding component type. System modelling in CHESS includes support for contract-based design, which relies on describing behaviours of components in terms of contracts. CHESS supports modelling of both strong and weak contracts and their association with components and system requirements. Moreover, *delegationConstraint* modelling element can be used to instantiate a component parameter in the given system model. Furthermore, CHESS facilitates interfacing with OCRA, such that the CHESS model together with the contracts is exported in the Othello System Specification (OSS) format used by OCRA. The contract checking is done by OCRA and the result is back-propagated to the CHESS model, as shown in Fig. 1.

Contract-Based Design: OCRA [7] is a tool for compositional verification of logic-based contract refinement built upon the OSS language, supporting a trace-based approach to contract based design. The semantics of both components and contracts is built around the notion of a *trace*, i.e., the observable part of an execution of a component. Following the trace-based semantics, a *component* S is described with a set V_S of variables that are visible outside of the component, and a set of all traces over V_S is denoted as $Tr(V_S)$. Then, an environment of S is a subset of $Tr(V_S)$. Assuming an assertion language, an assertion A can be described by an associated set of ports V_A and a semantics $[\![A]\!]$ defined as a subset of $Tr(V_A)$. Building on top of the assertion language, a contract C= (A, G) of the component S is a pair of assertions namely assumptions (A) and guarantees (G) over V_S. An environment E is said to be a correct environment of C iff $E \subseteq [\![A]\!]$. Contract *refinement* represents the backbone of checking the component decomposition [7]. Informally, a set of contracts of the sub-components *refines* a contract of the composite component if: (i) the assumptions of all sub-component contracts are met by the other sub-components and the environment defined by the assumptions of the composite component contract; and (ii) the sub-component contracts deployed in the environment defined by the composite contract assumptions imply the composite contract guarantees. For a formal definition of the refinement refer to [7].

Safety Case Modelling: A safety assurance case is often defined as an explained and well-founded (supported by evidence) structured argument to show that the system is acceptably safe to operate in a given context [12]. It is often required (explicitly or implicitly) by safety standards such as ISO 26262. Safety case is composed of all the work products gathered during the development of a safety-critical system. The spine of a safety case is a safety argument which connects the safety requirements and the evidence supporting and justifying those requirements. Goal Structuring Notation (GSN) [12] is a graphical argumentation notation used for safety case modelling. Since similar rationales exist behind specific arguments in different contexts, argument patterns of reusable reasoning are defined by generalising the specific details of an argument. The basic elements of GSN are shown in Fig. 3, for more details we refer the reader to the GSN Standard document [12]. To provide a better portability and exchange of the safety arguments, a Structured Assurance Case Meta-model (SACM) [13] standard is developed by Object Management Group. Since SACM 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 and Claims-Arguments-Evidence (CAE).

OpenCert is an assurance and certification tool environment with a safety argumentation modelling editor compliant with the standardised SACM. It further includes a Connected Data Objects[5] (CDO) server that supports collaborative modelling. In particular, it stores the safety case models in a database on a CDO server such that different distributed clients can access the models and work on the same safety case concurrently.

2.2 The Motivating Case

In this paper we will use a wheel-loader use case [4] to validate our approach. Wheel-loaders are usually equipped with a loading arm, which can perform up and down movements. The Loading Arm Control Unit (LACU) is the software control unit that coordinates the arm movement. The LACU architecture modelled in CHESS is shown in Fig. 2. It consist of a component providing the current

[5] https://www.eclipse.org/cdo/

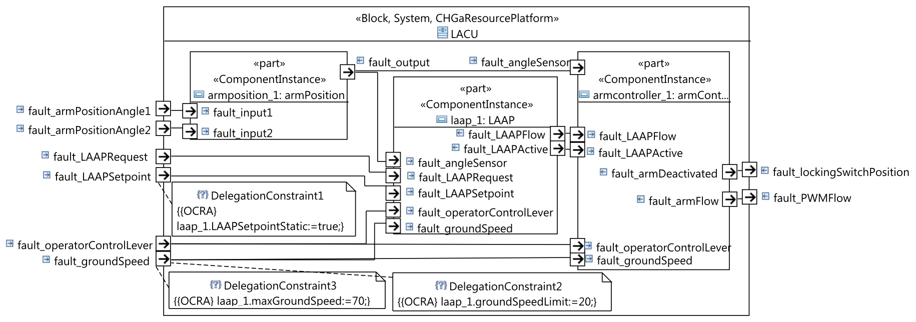


Fig. 2. The CHESS diagram of the LACU architecture

arm position, and an arm controller which sends the arm movement command. Moreover, it includes the Loading Arm Automatic Positioning (LAAP) component which can automatically move the arm to a pre-defined position. In this particular LACU the position is fixed (whereas it in other cases can be modified by the operator), while the maximum ground speed of the vehicle is 70km/h and the speed limit for moving the arm is 20 km/h, as shown in Fig. 2. The LAAP component is developed independently of this system as a SEooC.

The LACU safety analysis revealed the following system hazards: (1) unintended arm movement, and (2) arm movement during high speed (i.e., when the maximum speed of is greater than the ground speed limit). Some of the safety requirements defined to minimise the risks of those hazards from occurring are SR1:"*The stop position of the loading arm shall not deviate more than +-0.04 rad*" and SR2:"*The loading arm shall be disabled during high speed*".

3 Contract-Driven Assurance and Reuse

In this section, we present the methodology for supporting contract-driven assurance and reuse of safety relevant components. We first describe how to assure safety requirements validated through contract-based design. Then, we focus on the support for the contract-driven reuse of the components and their assurance information in the context of a trace-based approach to contract-based design.

3.1 Contract-Driven Assurance

To assure that a system such as LACU satisfies a given safety requirement based on the related contract, we need to provide evidence that the contract correctly represents the requirement (often said that its guarantees formalise the requirement) and evidence that the contract is satisfied with sufficient confidence in the given system context. We refer to this argument strategy as the contract-based requirements satisfaction pattern, shown in Fig. 3.

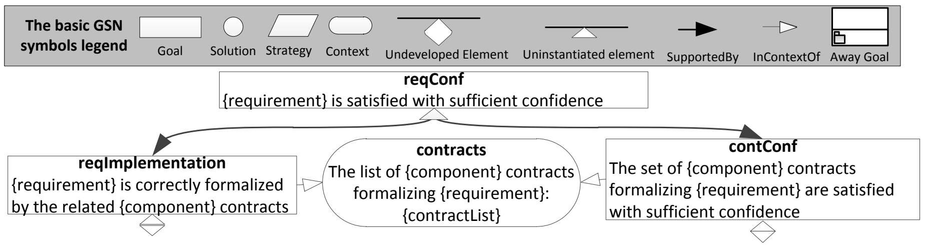

The basic GSN symbols legend
Goal
Solution
Strategy
Context
Undeveloped Element
Uninstantiated element
SupportedBy
InContextOf
Away Goal
reqConf
{requirement} is satisfied with sufficient confidence
reqImplementation
{requirement} is correctly formalized by the related {component} contracts
contracts
The list of {component} contracts formalizing {requirement}: {contractList}
contConf
The set of {component} contracts formalizing {requirement} are satisfied with sufficient confidence

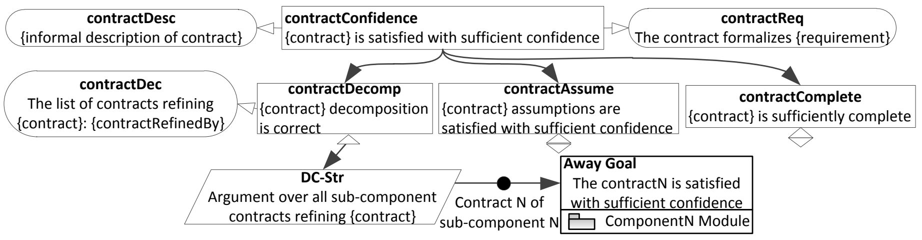
Fig. 3. Contract-driven requirement satisfaction assurance argument pattern

While compositional verification of a system using contracts establishes validity of a particular requirement on the system model in terms of contracts, confidence that the system implementation actually behaves according to the contracts should also be assured. Hence, to drive the system assurance using contracts we have associated assurance assets with each contract. Those assets can be different kinds of evidence that increase confidence that the component (i.e., the implementation of the contracts) behaves according to the contract, i.e., that the component deployed in any environment that satisfies the contract assumptions exhibits the behaviours specified in the corresponding contract guarantees. To argue that a contract is satisfied with sufficient confidence we need to assure that the component actually behaves according to the contract, and that the environment in which the component is deployed satisfies the contract assumptions [14]. But when we deal with hierarchical systems where contracts are defined on each hierarchical level with well defined decomposition conditions, then to argue that the composite component behaves according to the contract, we should explicitly argue over the component decomposition. The argument-pattern in Fig. 4 presents an extended contract-satisfaction argument pattern [14] with contract decomposition.

The extension assures that for each of the contracts on the composite component level (e.g., LACU) related to the requirement we are assuring, we should ensure that we have confidence in the component decomposition described by the refinement relationship (the *contractDecomp* goal). The goal is decomposed such that we argue over confidence in all subcomponent contracts specified through the refinement relationship. While the *contractDecomp* goal assures that what the component offers is supported by the confidence in the internal subcomponent specification, the *contractAssume* goal assures that the environment of the component/system meets the relevant assumptions.

3.2 Contract-Driven Reuse of Safety-Relevant Components

Reuse is intrinsic to contract-based design. It enables checking if a component can be reused in a particular system, i.e., whether the system meets its demands and whether the component meets the demands of the system. The support for reuse in contract-based design has been mainly focused on components (i.e.,



contractDesc
{informal description of contract}
contractConfidence
{contract} is satisfied with sufficient confidence
contractReq
The contract formalizes {requirement}
contractDec
The list of contracts refining {contract}: {contractRefinedBy}
contractDecomp
{contract} decomposition is correct
contractAssume
{contract} assumptions are satisfied with sufficient confidence
contractComplete
{contract} is sufficiently complete
DC-Str
Argument over all sub-component contracts refining {contract}
Contract N of sub-component N
Away Goal
The contractN is satisfied with sufficient confidence
ComponentN Module

Fig. 4. Contract satisfaction assurance argument pattern

implementations of contracts) and not reusable components as implementations of a set of contracts for different environments that may or may not be satisfiable together. As mentioned in Section 1, we refer to contracts that are not required to be satisfied by all correct environments as weak contracts. Conversely, the strong contracts define all the correct environments, i.e., all correct environments need to comply with all the strong contracts, while typically only some correct environments need to comply with a particular weak contract.

We formally describe the strong and week contracts in terms of environments in the context of the trace-based contract framework [7]: for a component S described with a set of strong contracts $\xi_S(S)$ and a set of weak contracts $\xi_W(S)$, we say that an environment E is a *correct environment* of S if: $\forall (A,G) \in \xi_S(S), E \subseteq [\![A]\!]$, i.e., for an environment of S to be correct, it must satisfy the assumptions of all the strong contract of S. We denote with $\mathcal{E}(S)$ all the correct environments of S. Such correct environments may or may not satisfy the assumptions of the weak contracts of S. While this provides some flexibility in specification of contracts, it may also mean that some weak contracts may never be validated in any of the correct environments e.g., if a weak contract is contradicting a strong contract. For S not to contain such unnecessary weak contracts we require that each weak contract of S has at least one correct environment that satisfies its assumptions, i.e.,: $\forall (A,G) \in \xi_W(S), \exists E \in \mathcal{E}(S), E \subseteq [\![A]\!]$.

The problem with specifying such contracts is that if we try to check refinement by considering all the specified weak contracts, the check will fail since a single environment might not be able to meet the assumptions of all the weak contracts. To overcome this problem without redefining the notion of contract refinement, we can either (i) filter the weak contracts before checking the refinement, such that only weak contracts whose assumptions are met by the current environment are included in the refinement check; or (ii) transform the weak contracts in a different format such that refinement can be performed:

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

Weak Contract Transformation: Instead of filtering only some weak contract to perform the refinement check, the refinement check could be performed if the weak contracts are transformed such that they do not impose restrictions on the environment. This can be done if the weak contract assumptions are relaxed. For a weak contract $C = (A, G)$ of a component S, a relaxed counterpart of this weak contract would be $C' = (\mathit{true}; A \implies G)$, where *true* represents an assertion satisfied by all environments. The relaxed counterpart has relaxed assumptions, hence it differs from the corresponding weak contract in terms of environments, but they are the same from the perspective of implementations. Since the assumption of C' is satisfied by every correct environment of S, it can be regarded as a strong contract. Since any contract that is refined by C is also refined by C', either form can be used for the sake of checking refinement of a weak contract. If we have a set of weak contracts and we transform them to their relaxed form and conjunct them to a single contract by conjunction of their guarantees, then any contract that is refined by at least one of those weak contracts is also refined by the conjuncted contract. The SEooC instantiation in a particular context does not require contract filtering in this case, but the in-context component can inherit both strong and weak contracts. Since the refinement check by considering all the strong and weak contracts would fail in case of two weak contracts that do not share the same correct environments, we transform the weak contracts to the appropriate format described as follows: given a SEooC component S and its instantiation S' in a correct environment $E \in \mathcal{E}(S)$, the set of contracts of S' denoted with $\xi(S')$, which contains the contracts considered during refinement check, is a union of all the strong contracts from $\xi_S(S)$ and the conjuncted contract of all the weak contracts in their relaxed form from $\xi_W(S)$.

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

3.3 Tool Support

We build upon the synergy of the three tools presented in Section 2 and implement the contract-driven assurance and reuse methodology by developing new and upgrading the existing plugins within the tools. We extend CHESS to support SEooCMM by adding the possibility to capture information about assurance assets and their relation to the corresponding contracts. With OCRA results back-propagated to the CHESS model, we perform automated weak contract filtering for the component instances. Upon updating the CHESS model, we then automatically instantiate the contract-driven assurance argumentation patterns for each component in the CHESS model. The generated argumentation is stored on a CDO server which can be accessed by any OpenCert argumentation editor connected to the CDO server. In the reminder of the section we detail the implementation (available on the CHESS [6] and OpenCert[7] repositories) of refinement checking with strong and weak contracts as an extension of CHESS and the automatic argument generation as an OpenCert plugin.

Refinement Checking With Strong and Weak Contracts: As mentioned in Section 3.2, to use a contract checking engine such as OCRA, which does not distinguish between strong and weak contracts, we can either support "weak contract filtering" as a part of reusable component instantiation or weak contract transformation to an appropriate format. We extend CHESS so that we can check all the weak contract validity and automatically update the component instance by indicating which weak contracts are valid in the given environment.

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

– *org.polarsys.chess.contracts.transformations* – contains model to text [15] transformation for generating the .oss file representing the model;
– *org.polarsys.chess.contracts.integration* – contains interface for communicating with OCRA.

Automated Argument-Fragment Generation: To facilitate automated instantiation of the contract-driven assurance pattern from Section 3, we implement the *ArgumentGenerator* plugin[8] within OpenCert. The user is prompted to select both the source CHESS model and the target assurance case in the CDO repository. The plugin generates a set of argument-fragments from the source CHESS model and stores them in the corresponding target assurance case in

[6] https://git.polarsys.org/c/chess
[7] https://git.polarsys.org/c/opencert
[8] org.eclipse.opencert.chess.argumentGenerator

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

4 LACU Case Study

In this section, we present our case study with the objective to apply the tool-supported contract-driven assurance and reuse methodology on a real-world case and evaluate its adequacy for automated support of assurance and reuse of assurance assets. We first present the failure propagation modelling in CHESS of the LACU and its in-context components, as well as the reusable LAAP component. Then, we discuss the contract checking results, and present the automatically generated argument-fragments.

4.1 Failure Propagation Modelling

To analyse the satisfaction of the safety requirement SR1 mentioned in Section 2.2, we model LACU with faults as different input/output ports of the components. For example, we consider the deviation of +-0.04 rad from the stop position to be a fault of the LACU arm positioning command represented by the *fault_PWMFlow* port. Hence, the goal of the contract corresponding to such an interpretation of SR1 would be to guarantee that *fault_PWMFlow* never occurs. To guarantee such a property in the context of the LACU defined by the parameters specified in Fig. 2, both one of the angle sensors and the ground speed sensor need to provide correct values. Furthermore, the operator inputs LAAPRequest and operatorControlLever should be fault free as well. This is captured in the *LACU_fault_propagation* contract in Fig. 5. To ensure that the component decomposition with respect to the fault propagation is done correctly, we define fault propagation contracts on the sub components as well. Fig. 5 presents the contracts for LACU, and its armPositioning and armController sub-components, as these are the components modelled for the particular wheel-loader.

The contracts of LAAP as a reusable component are specified separately, as they deal with not just this particular wheel-loader, but also other wheel-loaders that may support dynamic automatic positioning or that may or may not be able to move at high speed. We define four different weak contracts for the four different environments based on the two aspects of the wheel-loaders: dynamic automatic positioning and high-speed capability. In all environments the LAAP depends on fault-free user input, hence all contracts have the same assumptions considering fault free LAAPRequest and operatorControlLever. But on top of those conditions, for LAAP to ensure it will not issue a faulty arm positioning

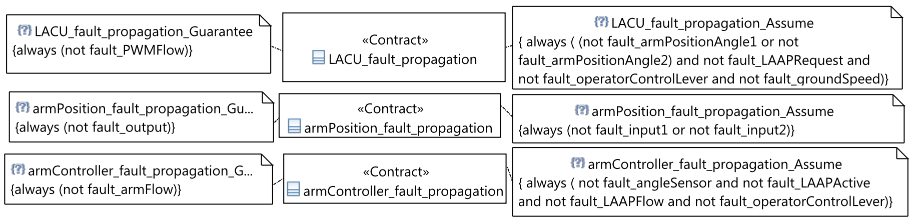

LACU_fault_propagation_Guarantee
{always (not fault_PWMFlow)}
«Contract»
LACU_fault_propagation
LACU_fault_propagation_Assume
{ always ((not fault_armPositionAngle1 or not fault_armPositionAngle2) and not fault_LAAPRequest and not fault_operatorControlLever and not fault_groundSpeed)}
armPosition_fault_propagation_Gu...
{always (not fault_output)}
«Contract»
armPosition_fault_propagation
armPosition_fault_propagation_Assume
{always (not fault_input1 or not fault_input2)}
armController_fault_propagation_G...
{always (not fault_armFlow)}
«Contract»
armController_fault_propagation
armController_fault_propagation_Assume
{ always (not fault_angleSensor and not fault_LAAPActive and not fault_LAAPFlow and not fault_operatorControlLever)}

Fig. 5. The LACU strong contracts specified in CHESS

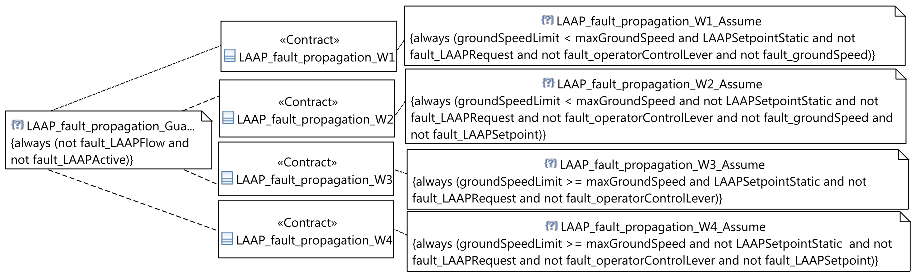

LAAP_fault_propagation_Gua...
{always (not fault_LAAPFlow and not fault_LAAPActive)}
«Contract»
LAAP_fault_propagation_W1
LAAP_fault_propagation_W1_Assume
{always (groundSpeedLimit < maxGroundSpeed and LAAPSetpointStatic and not fault_LAAPRequest and not fault_operatorControlLever and not fault_groundSpeed)}
«Contract»
LAAP_fault_propagation_W2
LAAP_fault_propagation_W2_Assume
{always (groundSpeedLimit < maxGroundSpeed and not LAAPSetpointStatic and not fault_LAAPRequest and not fault_operatorControlLever and not fault_groundSpeed and not fault_LAAPSetpoint)}
«Contract»
LAAP_fault_propagation_W3
LAAP_fault_propagation_W3_Assume
{always (groundSpeedLimit >= maxGroundSpeed and LAAPSetpointStatic and not fault_LAAPRequest and not fault_operatorControlLever)}
«Contract»
LAAP_fault_propagation_W4
LAAP_fault_propagation_W4_Assume
{always (groundSpeedLimit >= maxGroundSpeed and not LAAPSetpointStatic and not fault_LAAPRequest and not fault_operatorControlLever and not fault_LAAPSetpoint)}

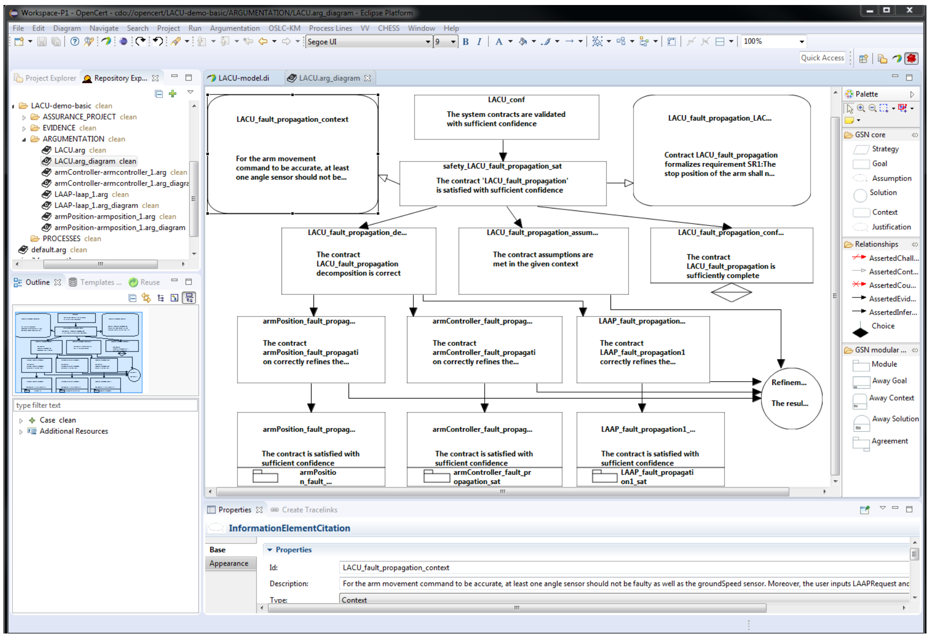
Fig. 6. LAAP fault propagation weak contracts specified in CHESS

command in each of the four environments it needs additional conditions to be met, sometimes stronger and sometimes weaker. The LAAP weak contracts for the four contexts modelled[9] in CHESS are shown in Fig. 6. In particular, for the LAAPFlow not to be faulty when the wheel-loader is capable of high speed and has a static automatic positioning setpoint, the only additional assumption on the environment is that the ground speed sensor is not faulty, as captured by the *LAAP_fault_propagation_W1* contract. On the other hand, when in addition to the high speed capability, the setpoint is dynamic, then the LAAP component requires not only ground sensor to be fault free, but also the LAAP setpoint value to be correct (the *LAAP_fault_propagation_W2* contract). Conversely, when the vehicle is not capable of high-speed and when the setpoint is static, then LAAP has no additional constraints (the *LAAP_fault_propagation_W3* contract). Finally, when the vehicle is not capable of high speed and the setpoint is not static, then the only additional constraint is on the correctness of the LAAP setpoint value (the *LAAP_fault_propagation_W4* contract).

4.2 LACU Assurance

To assure SR1 related to the fault propagation contracts of LACU, we first validate the weak contracts and then perform a refinement check. The weak contract validity check identifies that only the *LAAP_fault_propagation_W1* weak contract

[9] The contract type information is attached to the component and not shown here.


LACU_fault_propagation_context
For the arm movement command to be accurate, at least one angle sensor should not be...
LACU_conf
The system contracts are validated with sufficient confidence
LACU_fault_propagation_LAC...
Contract LACU_fault_propagation formalizes requirement SR1:The stop position of the arm shall n...
safety_LACU_fault_propagation_sat
The contract 'LACU_fault_propagation' is satisfied with sufficient confidence
LACU_fault_propagation_de...
The contract LACU_fault_propagation decomposition is correct
LACU_fault_propagation_assum...
The contract assumptions are met in the given context
LACU_fault_propagation_conf...
The contract LACU_fault_propagation is sufficiently complete
armPosition_fault_propag...
The contract armPosition_fault_propagati on correctly refines the...
armController_fault_propag...
The contract armController_fault_propagati on correctly refines the...
LAAP_fault_propagation...
The contract LAAP_fault_propagation1 correctly refines the...
Refinem...
The resul...
armPosition_fault_propag...
The contract is satisfied with sufficient confidence
armPositio n_fault_...
armController_fault_propag...
The contract is satisfied with sufficient confidence
armController_fault_pr opagation_sat
LAAP_fault_propagation1_...
The contract is satisfied with sufficient confidence
LAAP_fault_propagati on1_sat
LACU-model.di
LACU.arg_diagram
Palette
GSN core
Strategy
Goal
Assumption
Solution
Context
Justification
Relationships
AssertedChall...
AssertedCont...
AssertedCou...
AssertedEvid...
AssertedInfer...
Choice
GSN modular ...
Module
Away Goal
Away Context
Away Solution
Agreement
Project Explorer
Repository Exp...
LACU-demo-basic
ASSURANCE_PROJECT
EVIDENCE
ARGUMENTATION
LACU.arg
LACU.arg_diagram
armController-armcontroller_1.arg
armController-armcontroller_1.arg_diagram
LAAP-laap_1.arg
LAAP-laap_1.arg_diagram
armPosition-armposition_1.arg
armPosition-armposition_1.arg_diagram
PROCESSES
default.arg
Outline
Templates ...
Reuse
Case
Additional Resources
Properties
Create Tracelinks
Base
Appearance

InformationElementCitation

Properties

Id: LACU_fault_propagation_context

Description: For the arm movement command to be accurate, at least one angle sensor should not be faulty as well as the groundSpeed sensor. Moreover, the user inputs LAAPRequest and

Type: Context

Fig. 7. Preview of the LACU automatically generated argument-fragment in OpenCert

is valid in the given LACU context. Hence, only that contract is selected in the LAAP component instance. The informal description of each of the contracts is added to the CHESS model, as well as relations to the requirements. Once all the information is saved in the CHESS model and the status of the contracts is updated, we can proceed to automatically generate argument-fragments for each component in the system. Fig. 7 shows the screenshot of the Opencert interface presenting the result of the automatic instantiation of the contract-satisfaction argument pattern (Fig. 4) based on the information from the CHESS model of LACU. The list of automatically generated argument-fragment diagrams for each LACU component is in the top-left corner of the OpenCert interface.

4.3 Discussion

Contract-based design inherently supports reuse of components in form of contract implementations. But to fully understand the behaviour of a component and its safety implications, the context in which that behaviour is exhibited needs to be known. While component contracts represent a way of capturing a part of that context, additional context information is typically needed when dealing with safety-relevant components. In this case study, we have demonstrated how contract-based design can support reuse beyond implementations, to also include safety assurance artefacts related to those implementations.

Whether we perform the development of safety-relevant components in- or out-of-context, for reuse or just for a single system, different stakeholders are usually involved in the development process. For example, the expert performing the contract specification in a formal specification language such as OSS is not necessarily the same stakeholder as the one performing assurance modelling. Adoption of contracts just as any other formal specification is often hindered by the fact that not everyone can master a formal language [16]. Hence, for the stakeholder performing assurance modelling that should build upon different contract checks, we deem it is useful to accompany the contracts with additional information by the stakeholder that actually specified the contracts. Furthermore, a potential verifier assigned to verify certain behaviours of the component specified in a contract can directly associate that evidence with the contract and describe the results. While the goal of the LACU case study was not to evaluate the influence of our methodology on the quality of communication between different stakeholders in the development, during testing of the AMASS platform and collaborating on both modelling and assuring different systems, we could experience some of the communication benefits. Capturing all the safety assurance relevant information provided by different stakeholders in safety-critical system development in a traceable way has the potential of enhancing the collaboration between different stakeholders in building an assurance case. Moreover, by automatically generating parts of the argumentation, the safety engineer gets a head-start in assuring the system safety.

5 Conclusions and Future Work

Reuse of safety-relevant components in safety-critical systems needs to cover more than just the implementation. Enriching contract-based design by associating contracts with assurance information enables us to reuse assurance artefacts together with the accompanying contract implementations. Furthermore, enabling variability modelling of the contract specifications in terms of strong and weak contracts allows us to provide greater support for reuse of components explicitly developed for reuse in different contexts. We have presented a tool support for the methodology by introducing system modelling with strong and weak contracts and their alignment with trace-based contract-based design. Furthermore, we have enabled automatic instantiation of assurance argument-fragments from the enriched system models. The presented tool support and the case study illustrate the feasibility of our contract-driven assurance and reuse methodology to assist in assuring requirements satisfaction and reusing assurance information.

To reap the full benefits of contract-driven assurance and reuse, further extensions to the AMASS platform are needed. Extending the underlying meta-model to connect the contracts with component failure behaviour could enable instantiation of many argument patterns that focus on failure behaviour. Furthermore, the traceability between the system and assurance modelling achieved through the contracts could be further enriched to support analysis and assurance of the interplay of multiple system concerns such as safety and security.

Acknowledgements. This work is supported by the EU and VINNOVA via the ECSEL Joint Undertaking projects AMASS (No 692474) and SafeCop (No 692529), as well as the Swedish Foundation for Strategic Research (SSF) via the FiC project.

References

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

[2] J.-M. Jézéquel and B. Meyer. Design by Contract: The Lessons of Ariane. *IEEE Computer*, 30(1):129–130, January 1997.

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

[4] Irfan Sljivo, Barbara Gallina, Jan Carlson, Hans Hansson, and Stefano Puri. A method to generate reusable safety case argument-fragments from compositional safety analysis. *Journal of Systems and Software: Special Issue on Software Reuse*, July 2016.

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

[6] I. Sljivo, B. Gallina, J. Carlson, and H. Hansson. Strong and Weak Contract Formalism for Third-Party Component Reuse. In *3rd International Workshop on Software Certification*, pages 359–364. IEEE, November 2013.

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

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

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

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

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

[12] Goal Structuring Notation Working Group. GSN Community Standard V1.0. Origin Consulting (York) Limited, 2011.

[13] Object Management Group. SACM: Structured Assurance Case Metamodel. Technical report, V1.0. http://www.omg.org/spec/SACM, 2013.

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

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

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