Guiding Assurance of Architectural Design Patterns for Critical Applications

Irfan Sljivo[1], Garazi Juez Uriagereka[2], Stefano Puri[3], and Barbara Gallina[1]

[1] Mälardalen University, Västerås, Sweden
{irfan.sljivo,barbara.gallina}@mdh.se
[2] ICT Division, TECNALIA, Derio, Spain garazi.juez@tecnalia.com
[3] Intecs, SpA, Pisa, Italy stefano.puri@intecs.it

Abstract. Development of critical systems nowadays is hardly achievable without reuse of previous knowledge. Design patterns have an important role in the design of such systems as they define and document common solutions to recurring design problems. However, critical systems such as those that are safety or security related, often require specific assurances that the system is adequate to operate in a given environment. Just as with any other reused knowledge in such systems, the reuse via application of design patterns needs to be assured every time. In this paper, we present a methodology for assuring the application of design patterns in critical domains. In particular, we enrich the design patterns template to support their further assurance. We define the aspects that should be tackled during the assurance of a design pattern application. We use the information specified in the design pattern template to guide the automated instantiation of the argumentation for each design pattern application in the system. We provide tool-support for our methodology in the context of the AMASS tool-platform and evaluate it in an automotive case study.

1 Introduction

Many critical systems nowadays have become so complex that they are rarely built from scratch. Instead, pre-developed components as well as the knowledge about addressing particular problems are reused from previous experience. Design patterns were proposed to establish a common solution to recurring design problems occurring in communities and buildings [1] as well as in software and systems design [2]. Design patterns facilitate the system architect's work when facing commonly recurring design problems. However, when design patterns are used in critical systems that require specific assurances regarding adequateness to operate in a given environment, the application of a particular design pattern needs to be accompanied by evidence supporting its usage. That evidence is often referred within an assurance case argument, which connects the evidence with the to-be-met system-related requirements (claims) [3]. While some of the evidence supporting a design pattern could be reusable together with the pattern, there are certain aspects of assurance of a design pattern that are specific

to a particular system and cannot be simply reused. Assurance over the use of a design pattern in a particular system needs to take into account whether the pattern is applicable, or whether it is the right choice, to solve the particular problem for which it is used in that system. Furthermore, even if the pattern is the right choice to solve a particular problem, the question whether the pattern is correctly applied still remains.

Contract-based design aims to support reuse and independent development of components by using contractual specifications to check the compatibility of the independently developed parts [4]. A contract is defined as a pair of assertions (i.e., assumptions) and guarantees (i.e., properties) such that the guarantees hold only if the assumptions also hold. Formal and semi-formal contracts facilitate automated analysis, while informal contracts are useful for manual reviews of component compatibility. We see contracts in general as very useful for reuse of design patterns. Having a contract, regardless of the level of formality, associated to a specific architectural pattern allows reuse of some of the associated assurance information and, even more importantly, it can support automated generation of some of the system-specific assurance argument fragments related to the design pattern.

In this paper, we present our methodology for assurance of design patterns using contracts. In particular, we propose an extended design pattern template to include the contract's assumptions and guarantees regarding the design pattern, as well as additional information that is needed for the assurance of the use of the design pattern in a critical system such as a car or an aircraft. We provide tool support for defining, storing and applying design patterns with the associated template that currently includes specification of informal contracts. Building on top of the information from the extended design pattern template and on top of previous work regarding generation of arguments from contract-based specification [5], we present an assurance argument pattern to guide the design pattern assurance. The argument pattern highlights the different aspects relevant for assuring the usage of a design pattern in a critical system. We present the tool-support for the methodology in the AMASS platform, which we use to evaluate the proposed methodology in an automotive case study considering both safety and security related design patterns.

The rest of this paper is organised as follows: in Section 2, we present the essential background information. In Section 3.1, we present our design pattern assurance methodology. In Section 4, we present an application of the proposed methodology onto an automotive case study. In Section 5, we present the related work. Finally, in Section 6, we draw our conclusions and outline future work.

2 Background

2.1 Assurance Case Argument Representation

An **assurance case** is defined as *"a collection of auditable claims, arguments, and evidence created to support the contention that a defined system/service*

will satisfy its assurance requirements." [3]. A **claim** is defined as *"a proposition being asserted by the author or utterer that is a true or false statement"* [3], while an **argument** is defined as *"a body of information presented with the intention to establish one or more claims through the presentation of related supporting claims, evidence, and contextual information"* [3].

The assurance case argument can be documented in different ways ranging from free text to (semi-)formal textual or graphical notations. In this work we use Goal Structuring Notation (GSN) [6] to represent the assurance case argument. GSN is a graphical argumentation notation, which can be used to document different types of arguments. The principal elements of GSN are shown in Fig. 1. The main purpose of GSN is to show how **goals** (claims about the system), are broken down into **subgoals** and supported by **solutions** (the gathered evidence used to back up the claims). The argument elements can be connected with one of the two relationships: *supportedBy* and *inContextOf*. The **supportedBy** relationship is used to connect goals and strategies with other subgoals, strategies and solutions, while the **inContextOf** relationship is used to connect the goals and strategies with supporting elements such as contexts, justifications and assumptions. The modelling capabilities of GSN have been extended with structural and entity abstraction to support representation of patterns of reusable reasoning [6]. The right side in Fig. 1 presents some of the elements of the extended GSN. For structural abstraction the supportedBy relationship is extended by introducing multiplicity and optionality relationships. The **multiplicity** relationship indicates zero to many relationship between two elements, where n represents the cardinality of the connection. The **optionality** relationship indicates a zero or one cardinality connection between two elements.

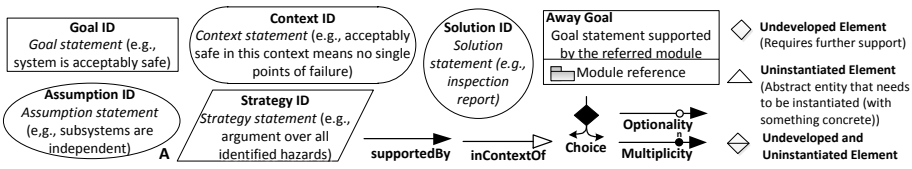

Fig. 1. Overview of the GSN elements

2.2 Architectural Design Pattern Representation

As mentioned in Section 1, design patterns are abstract representations for how to solve recurring design problems. Their main purpose is to help the architects and designers reuse existing solutions for design problem solving. The traditional design pattern definition included three aspects: context, problem, and solution [1]. These three aspects alongside the pattern name, constitute the traditional pattern representation. This pattern definition structure describes the basic relationships between the different aspects, e.g., the problem to be solved

in a particular context, the solution that successfully solves the problem in the particular context, and the context itself describing the preconditions that need to be satisfied for the solution to be successfully applied.

Besides this traditional design pattern definition with the basic four elements (name, context, problem, and solution), many other design patter definition templates have been proposed [7]. Some representations include information such as alternative names, related or similar patterns, motivations for using the pattern, its intent, etc. Besides the traditional parts of the design pattern definition, in this work we consider also information regarding the other known names of the pattern, its consequences, and implementations.

2.3 The AMASS OpenCert Platform

AMASS (Architecture-driven, Multi-concern and Seamless Assurance and Certification of Cyber-Physical Systems) [8] has created and consolidated the first de-facto European-wide open tool platform, ecosystem, and self-sustainable community for assurance and certification of Cyber-Physical Systems.

One of the main characteristics of the AMASS platform is its focus on seamless interoperability between assurance and engineering tasks of the system development. To achieve this, the AMASS tool platform [9] is created as an integration of different existing tools . The four core tools that constitute the AMASS tool platform are:

- **OpenCert**[4] for assurance case modelling and other certification-related activities such as evidence management and standards compliance checking.
- **The CHESS toolset**[5] for system modelling with support for model-driven component and contract-based development of high-integrity systems. The toolset is based upon the open source Papyrus UML editor[6].
- **EPF Composer**[7], recently migrated to Eclipse Neon 4.6.3 [10], for system and software process engineering including compliance management.
- **The BVR tool**[8] for variability management.

Besides the core tools, the AMASS tool platform is integrated with over 20 external tools. In this paper we focus on OpenCert and CHESS. On the one hand, the CHESS model-based methodology and toolset provide support for all phases of (sub)system design: from requirements definition, architecture modelling to software design and its deployment to hardware. Analysis, in particular dependability and performance, is also supported. On the other hand, OpenCert is an assurance and certification tool environment with a safety argumentation modelling editor compliant with the standardised Structured Assurance Case Metamodel (SACM) [3].

[4] https://www.polarsys.org/opencert/ (accessed June 1, 2019)
[5] https://www.polarsys.org/chess/ (accessed June 1, 2019)
[6] https://www.eclipse.org/papyrus/ (accessed June 1, 2019)
[7] https://www.eclipse.org/epf/ (accessed June 1, 2019)
[8] https://github.com/SINTEF-9012/bvr (accessed June 1, 2019)

3 Tool-Supported Architectural Design Pattern Assurance

The aim of this section is to present our methodology for assurance of design patterns with its essential building blocks, and realisation of the methodology in the AMASS platform.

3.1 Architectural Design Pattern Assurance Methodology

As discussed in Section 1, the main challenge with reuse using design patterns in critical applications is that applying design patterns brings in solutions designed outside of the context of the particular system. Consequently, the solution, devised elsewhere, needs to be assured as adequate for the particular critical system where it is being reused. This process of reuse and assurance of the to-bereused information is divided between steps performed outside of the context of a particular system, and those steps that have to be completed once the information is reused in the context of the particular system. With that division in mind, we define our methodology in the following steps:

- Out of context of a particular system:
  - Design pattern specification: This step is system independent and its aim is to capture the reusable knowledge in the design pattern template. The specified architectural patterns are stored in an architectural patterns library.
  - Arguments pattern specification: Just as with the design patterns, this step is system independent and aims to reuse the assurance strategies across systems that require an assurance case. Different argument patterns are specified for different architectural design techniques. In the case of design pattern assurance, we may specify multiple argumentation patterns for assurance of different design patterns. Such argument patterns are stored in an argument patterns library.
- In context of a particular system:
  - System modelling, which includes modelling of system components.
  - Design pattern selection based on matching the pattern contracts with the system requirements. The pattern is selected from the library and applied to a particular component in the system model.
  - System components refinement via integration of the selected design patterns in the system model. The inherited information from the design pattern, in particular the design pattern contracts, is used to check if the design pattern is compatible with the given system environment, i.e., whether the design pattern contract assumptions are met, and whether the corresponding guarantees are sufficient to meet the demands of the system environment.
  - Automated generation of assurance arguments for each design pattern application. For each design pattern application in the system model, a corresponding argumentation fragment is generated and used in the

assurance case. The argument fragment represents an instantiation of an argument pattern from the argument patterns library. In case there are multiple argument patterns specified for design pattern application, the user should be given the possibility to select which argument pattern to instantiate.

In the reminder of the section, we present the essential building blocks of the methodology. In particular, we present the design pattern template enriched with assurance related information, the assurance argument pattern to guide the assurance of a particular design pattern in a system, and the tool support for the presented methodology in the AMASS platform.

Design pattern template As mentioned in Section 2.2, applying a design pattern in a critical application to solve a particular problem requires certain assurance to guarantee that the problem has been successfully addressed. This cannot be easily done with the current means (i.e., templates) for pattern definition. Thus, we propose an extension of design pattern templates to include design pattern contracts and guide the assurance of the application of design patterns in particular systems:

- **Pattern name**: define a name which describes the pattern univocally.
- **Other well-known names**: this item refers to other names with which the design pattern can be known in different domains of application or standards.
- **Intent/context**: define in which context the pattern is used. For example, define if the pattern is recommended for a specific safety-critical domain.
- **Problem**: description of the problem addressed by the design pattern.
- **Solution/Pattern Structure**: the solution to the problem under consideration. Main elements of the patterns are described.
- **Consequences**: define the implication or consequences of using this pattern. This section explains both the positive and negative implications of using the design pattern.
- **Implementation**: set of conditions that should be taken under consideration when implementing the pattern. Language dependent.
- **PatternAssumptions**: the design pattern contract assumptions.
- **PatternGuarantees**: the design pattern contract guarantees.

Using contracts for design patterns [11] offers a way of capturing under which conditions instantiating a pattern offers the desired specification. The PatternAssumptions and PatternGuarantees make the design pattern contract, which collects the information regarding the implication of using this pattern. In particular, the PatternAssumptions represent conditions that should be met for the correct usage of the design pattern, while the PatternGuarantees represent the conditions that the correct application of the pattern yields. Unlike in contract-based design, the design pattern contract in the current implementation is not specified in a formal or even semi-formal form, but rather as a set of informal statements that should be manually checked.

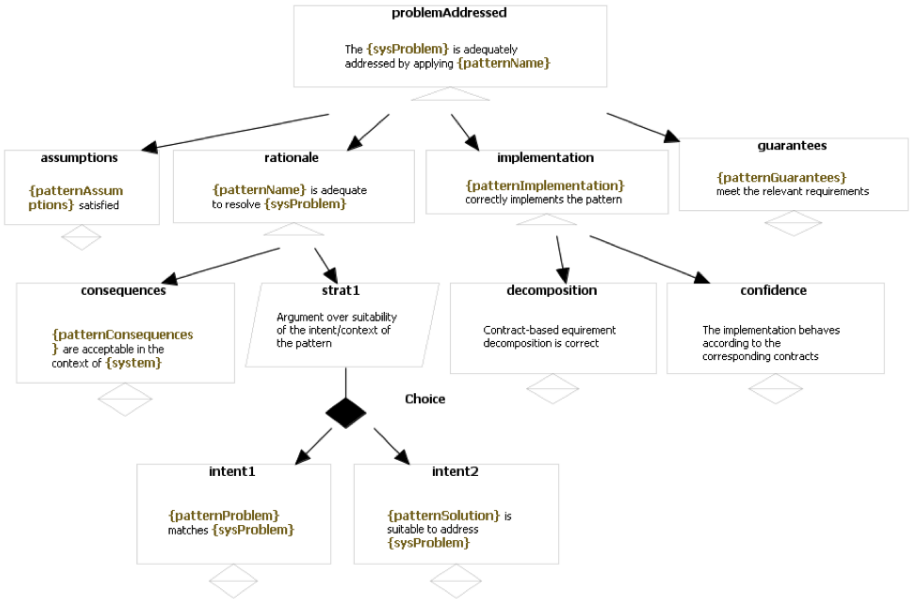

Fig. 2. High-level assurance argument-pattern for architectural pattern assurance

Assurance of design patterns Each design pattern offers a solution to a particular problem. The list of the known problems a pattern is addressing is captured in the design pattern template. After choosing to use a particular design pattern for potentially solving a specific problem, we need to assure that the pattern is suitable to address this problem as well as that the pattern has been correctly applied, according to the information from the design pattern template. An assurance argument pattern depicting the assurance of the application of an architectural pattern is presented in Fig. 2. We assure the suitability of a design pattern to the specific problem either by looking at the problem statement in the design pattern template and whether our problem matches any of the known problems the pattern is used for, or if our problem is not in that list, then we need to argue why is this pattern suitable to address that particular problem. Hence, we use the choice element in Fig. 2 when supporting the design pattern adequacy goal. Furthermore, the consequences of using the pattern should be acceptable in the context of the system. Once the pattern is deemed suitable, we then assure that the PatternAssumptions are met, and that the PatternGuarantees satisfy the relevant requirements. For example, introduction of an acceptance voting pattern influences timing behaviour of the system, hence we should assure that the PatternGuarantees do not impair the relevant timing requirements. Finally, we need to assure that the implementation of the pattern is performed correctly, i.e., that it conforms to the conditions specified in the design pattern template.

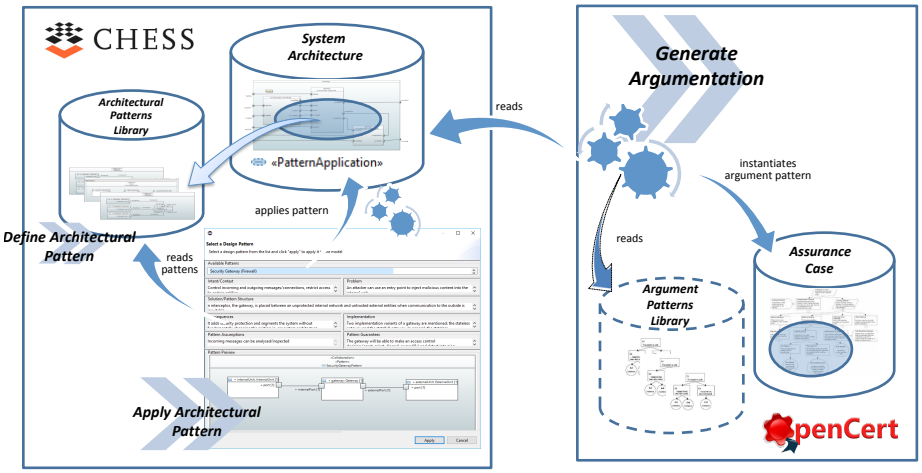

Fig. 3. Realisation of our methodology for design pattern application and assurance within the AMASS platform

The structure of the argument pattern complies with the ISO/IEC 15026 standard [12], which specifies the minimum requirements for the structure and contents of an assurance case. According to ISO/IEC 15026, the essence of an assurance caseis to capture the uncertainties regarding truth or falsehood of the claims we assure. Led by the same thought, the purpose of the argument pattern for assurance of the application of an architectural pattern is to highlight all the uncertainties involved in the application of the architectural pattern.

3.2 Tool Support in the AMASS platform

We have extended the AMASS platform to support the specification of design pattern templates and their application. Furthermore, we have automated the instantiation of the assurance argument-pattern presented in Fig. 2 from the information stored in the applied design pattern in the given system model. Fig. 3 depicts our methodology within the AMASS platform. In the out-of-context setting, design pattern specification is performed using CHESS and arguments pattern specification using OpenCert. In the in-context setting, CHESS is used for system modelling and design pattern instantiation. The component inherits the design pattern information through the PatternApplication relationship. Currently, we have extended CHESS to support only specification of informal design pattern contracts, so its contract-checking features cannot be used for design patterns, but design pattern contracts should be manually checked. Finally, automated generation of assurance arguments for each design pattern application is performed from the CHESS system model to the selected assurance case in OpenCert. In the reminder of the section we briefly present the tool support in some more detail.

Design pattern application and template specification: Patterns are modelled as a special kind of UML Collaboration [13], i.e. as a classifier owning a set of abstract collaborating entities called roles (CollaborationRoles); each role can have input and output ports and roles can be connected together via their ports. In particular, the Pattern stereotype extending the Collaboration UML base entity with the set of attributes foreseen by our pattern template has been defined.

In the CHESS environment, by using the Papyrus facilities, patterns can be defined in dedicated model libraries and then imported in a given model, for their application. The application of the pattern in a given model is performed by using the UML CollaborationUse construct; through the CollaborationUse it is possible to detail how the pattern described by a Collaboration is applied in a given context classifier (i.e. a system component), in particular by binding entities available in that context (i.e. sub-components, ports and connections) to the abstract entities defined for that particular pattern.

Without a specific tool support, the specification of the entire set of bindings required to properly model the instantiation of a collaboration/pattern in a given context can be an error-prone and hard task to be performed by the system architect. To solve this problem, in CHESS we implemented a dedicated wizard to easily allow the selection of the pattern to be instantiated in the context of a given system component and the specification of the mapping between the pattern roles/ports/connections and the actual entities available in the current model. The wizard allows to leave pattern related entities unbounded: in this case, pattern entities not mapped to system model entities are automatically created in the target model as new entities owned by the selected system component.

The pattern instantiation is stored in the target model, together with the information about the roles bindings; in this way it is always possible to query the model about the list of patterns that have been applied and validate each patterns instantiation by verifying the existence of the role bindings. Moreover, system components playing patterns roles are automatically stereotyped/tagged as PatternRole; a PatternRole comes with the information about the patterns in which the components plays a role (indeed in principle the same system component can contribute to the realization of different patterns), to let the system architect to easily retrieve patterns applications in the model and in the available diagrams.

Assurance argument pattern instantiation: To facilitate automated instantiation of the proposed argumentation pattern in Fig. 2, we implement the Argument Generator plugin within OpenCert and CHESS. The user is prompted to select both the source CHESS model and the target assurance case. The plugin generates a set of argument-fragments from the source CHESS model, one for each design pattern applied in the system model, and stores them in the corresponding target assurance case. The plugin assumes that the CHESS model contains the applied design patterns with their templates specified.

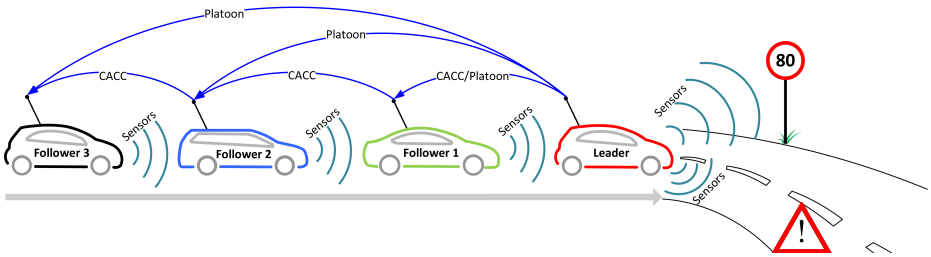

Fig. 4. Explanation of CACC and Platoon Driving

The argument patterns library for architectural assurance in OpenCert depicted with the dotted line in Fig. 3, is currently hardcoded and does not allow user the option of specifying multiple argument patterns and choosing which argument pattern to use for assuring a particular architectural design technique. For example, one predefined argument pattern is used for contract-based design, and another for application of design patterns.

4 Case Study

In this section we present our automotive case study. First, we describe the system that is the case of the study, its safety and security implications as well as the goal of the case study. Then, we describe the selected design patterns to address the specific safety and security requirements and specify them in the form of the design pattern template presented in Section 3.1. Next, we describe the pattern application performed on the system modelled in CHESS. Finally, we present the assurance of the selected patterns using the automated assurance argument support in the AMASS platform.

4.1 The Case of the Study: CACC and Car Platooning

Cooperative Adaptive Cruise Control (CACC) is a typical example of a cooperative safety-critical system where the cruise control of a vehicle is automatically guided by the information wirelessly received from the predecessor vehicle. Fig. 4 describes the different operation modes. The vehicle in CACC mode enhances the local sensor-based Adaptive Cruise Control (ACC) mode with a Car2Car link to the predecessor vehicle. Enhancing the CACC mode, in the platoon mode the vehicles use not only the information from the predecessor vehicle, but also from the leader of the platoon, which is usually the first vehicle in the string of vehicles. As a case study in the AMASS research project, we use a fleet of autonomous model cars that can run autonomously and sense the environment by means of camera and ultrasonic sensors [14]. Additionally, the cars can exchange Car2Car messages via WIFI connection and in that way operate in CACC and platoon modes.

The goal of the case study is to evaluate the feasibility of use of the AMASS tool-support design pattern application and assurance methodology to different kinds of design patterns. While we have initially developed the methodology for use in safety-critical domains, and in particular targeted the fault-tolerance design patterns, in this case study we aim to evaluate that the methodology can be also used for other types of design patterns such as those addressing security-related challenges.

Safety and security implications: Although there are different hazards that may be associated with the CACC and platooning modes, in this case study we focus on hazard *H1: inadequate longitudinal control* that may lead to rear collision of one or more vehicles. Given that the vehicles perform the longitudinal control autonomously in CACC and platooning modes, either based on the own sensory input or the input received from the other vehicles, to find causes to our hazard we turn to reliability and security of communication between the vehicles, as well as the reliability of the own sensory input. To address the hazard H1, we define the functional safety concept according to the functional safety standard ISO 26262 [15]. Although this standard cannot be directly applied to such system-of-systems such as CACC and car platooning, the basic concepts of defining the safety goals and the related functional safety requirements for each hazard are still applicable. A subset of the functional safety concept we defined to address the hazard is given in Table 1. To achieve the safety goal *SG1* we focus on the two particular functional safety requirements *FSR1* and *FSR2*. The first requirement aims to ensure that given a failure of any of the components on which the autonomous functionality depends on (be it in the own vehicle, in the communication channel or in the cooperating vehicle), the vehicle will go to the appropriate mode in order to increase system availability in light of failures. Implementing this requirement requires a monitoring component that issues a command to move the vehicle in the appropriate mode. While such monitoring component deals with the quality of communication and potential failures of components, it does not deal with the protection of the received and sent messages to ensure that they are coming from the correct source. For that purpose, we define FSR2 that is at the same time a security requirement. It enforces the communication between the vehicles to be encrypted, so that a random adversary cannot simply inject false messages in the communication between the platoon members.

4.2 The selected design pattern templates

Given that the problems in both of the functional safety requirements in Table 1 are quite common, we choose two particular design patterns to address each of them. For the first requirement we opt for the *Monitor-Actuator* design pattern (shown in Fig. 5 and described in [16]), while for the second requirement we opt for the *Security Gateway* design pattern (shown in Fig. 6 and described in [17]). We first bring the design patterns in the form of templates presented in Section 3.1, and then we apply those patterns in designing our CACC system.

Table 1. A subset of the functional safety concept

SG1:	A sudden braking manoeuvre of predecessor vehicle shall not result in distance shorter than 2m.
FSR1:	The vehicle shall maintain safe distance to the predecessor by monitoring the system and environmental conditions, and selecting an appropriate operation mode (platoon, CACC, ACC, manual) based on the observed conditions.
FSR2:	The Car2Car messages shall be encrypted to prevent against unauthorised messages propagating to the CACC control software.

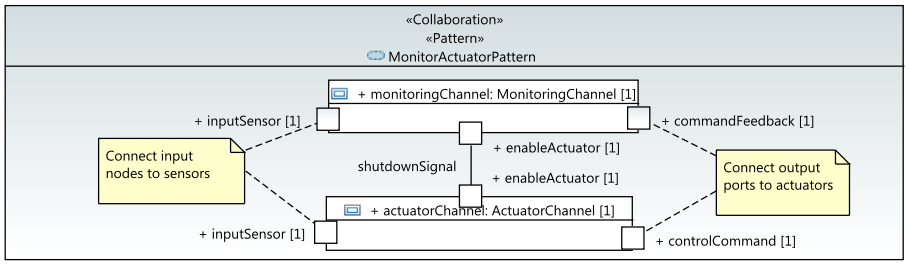

Fig. 5. The Monitor-Actuator Pattern modelled in CHESS

The Monitor-Actuator Pattern [16]:

- **Pattern name**: Monitor-Actuator Pattern.
- **Other well-known names**: -
- **Intent/context**: The Monitor-Actuator pattern is a special type of heterogeneous redundancy intended for systems with low availability requirements and a fail-safe system state.
- **Problem**: How to improve the safety of an embedded system in the presence of a single point of failure in a system that includes a fail-safe state and low availability requirements at reasonable cost.
- **Solution/Pattern Structure**: One channel is responsible for performing the end-to-end actuation, while the monitor channel compares the output of the actuator channel against the expected value and forces the actuation channel to enter a fail-safe state if a discrepancy is found. The two channels are independent such that faults from one are not affecting functioning of the other channel.
- **Consequences**: If a failure in the monitoring channel occurs, the actuation channel will continue to operate correctly unless there is a second failure in the actuation channel.
- **Implementation**: The monitor channel usually contains a reduced version of the actuation algorithm and is driven by a separate set of sensors. The same set of sensors may be used, but this introduces a common mode failure into the system. Latent failure must be avoided through periodic maintenance checks and/or specific built-in-tests.
- **PatternAssumptions**: -

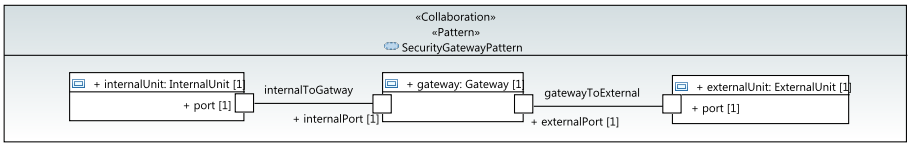

Fig. 6. The Security Gateway Pattern modelled in CHESS

- **PatternGuarantees**:
  1. There is no influence on the execution time of the actuation channel.
  2. The pattern does not improve the reliability of the system: $R_{new} = R_{old}$
  3. The percentage of the relative safety integrity improvement (RSI) is RSI $= R_{MC}C_{MC}$ x 100%, where R_{MC} is the reliability of the monitored channel, and C_{MC} is the probability that the failure in the actuation channel is detected by the monitoring channel.

The Security Gateway Pattern [17]:

- **Pattern name**: Security Gateway.
- **Other well-known names**: Firewall
- **Intent/context**: Control incoming and outgoing messages/connections, restrict access to certain entities.
- **Problem**: An attacker can use an entry point to inject malicious content into the internal unit.
- **Solution/Pattern Structure**:
  1. An interceptor, the gateway, is placed between an unprotected internal network and untrusted external entities when communication to the outside is inevitable.
  2. All incoming or outgoing messages pass through this host.
  3. The gateway controls the network access to the internal unit(s) according to predefined security policies and can also inspect msg content to detect intrusion attempts and anomalies.
- **Consequences**:
  1. It adds security protection and segments the system without fundamentally changing the existing in-car system architecture
  2. Security gateway might introduce latency into the communication, which is a subject of safety/performance impact analysis
- **Implementation**: Two implementation variants of a gateway are mentioned: the stateless gateway and the stateful gateway. In general, the stateless gateway is more performant and the statefull gateway is more advanced, i. e. it is able to make more intelligent decisions.
- **PatternAssumptions**: Incoming messages can be analysed/inspected
- **PatternGuarantees**: The gateway will be able to make an access control decision (grant, reject, discard, or modify) and detect intrusion attempts.

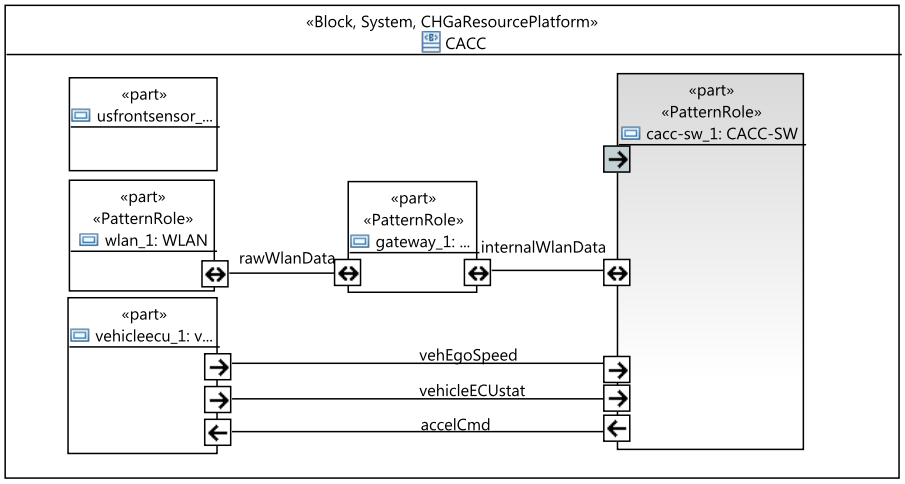

Fig. 7. The CACC system-level architecture with Security Gateway pattern application

4.3 Design pattern application

Fig. 7 presents the high-level architecture of the CACC system that shows the connection of the CACC software with the corresponding hardware elements, which include the front distance sensor, the WLAN adapter and the vehicle ECU that provides own vehicle information from other vehicle systems. We apply the security gateway pattern at this level, as it should add a security layer without fundamentally changing the existing in-car system architecture and it should handle all incoming and outgoing messages between the internal and external network. We place the gateway component between the WLAN adapter and the CACC software such that all communication between the autonomous functions of the vehicles in CACC and platoon modes goes through the gateway. The *cacc-sw* component (shown in Fig. 8) takes in the data from the gateway and the own hardware components, processes them and issues the longitudinal control command.

Fig. 9 represents the architecture of the CACC manager component as a part of cacc-sw in charge of computing and issuing the longitudinal control command. Since CACC manager contains the actuator component (*cacccontroller*), we choose to apply the Monitor-Actuator pattern in this subsystem. In particular, the pattern is applied to each system operating mode such that the fail safe for each mode represents transition to a lower mode of operation. We have the *caccstatemanager* component to act as the monitor, and the *cacccontroller* as the actuator. The two components implement the FSR1 from Table 1 such that *caccstatemanager* analyses the sensory and WLAN data input and sends a signal to the *cacccontroller* component if there is a failure detected and to which mode should the system transition.

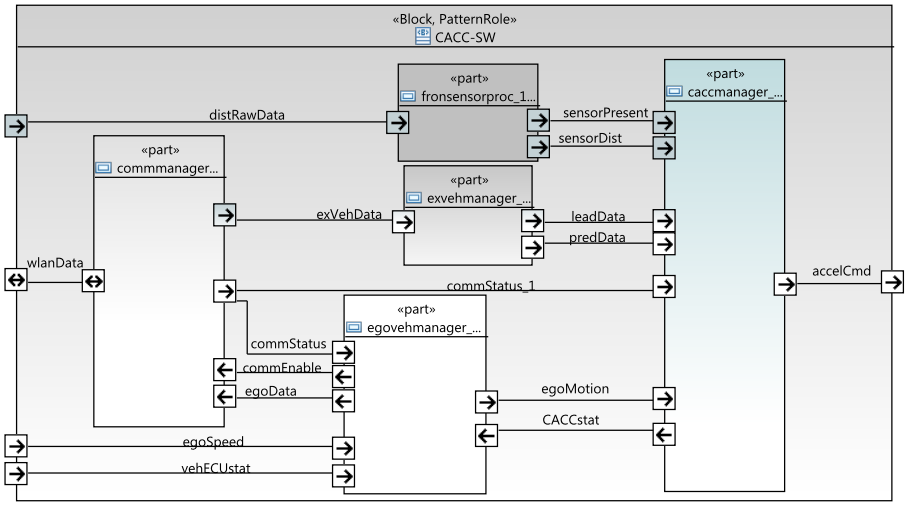

Fig. 8. The first-level of the CACC software architecture

4.4 Safety Assurance

We drive the assurance that the application of these patterns adequately addresses the corresponding requirements by considering the argument-pattern presented in Section 3.1. We discuss the assurance in according to the four first-level subgoals: pattern assumption satisfaction, the rationale, pattern implementation, and pattern guarantees adequacy.

Monitor-Actuator assurance: Since the design pattern template does not state any pattern assumptions, we do not further develop the goal. It implies that the system is not required to fulfil particular demands in order to enable application of this pattern. In the rationale argument branch we argue over the adequacy of the given pattern to resolve the problem at hand. Given the need to increase overall system availability by sacrificing availability in higher modes of operation upon failures, the pattern problem matches the system problem in this regard since the CACC operating modes do not have high availability requirements separately. Furthermore, the fact that the monitor component should not influence nominal behaviour of the system meets the needs of the system, so in that respect, the pattern consequences are acceptable within the CACC context. In the implementation branch of the argument we argue over adequacy of the particular instantiation of the pattern and that it correctly implements the design pattern. Since the implementation of the design pattern components can be described via semi-formal specifications, it is possible to use such specifications to argue over adequacy of the implementation. For example, the pattern template states in the implementation section that the monitor and the actuator may use the same or redundant inputs. As we can see from Fig. 8, in our case the

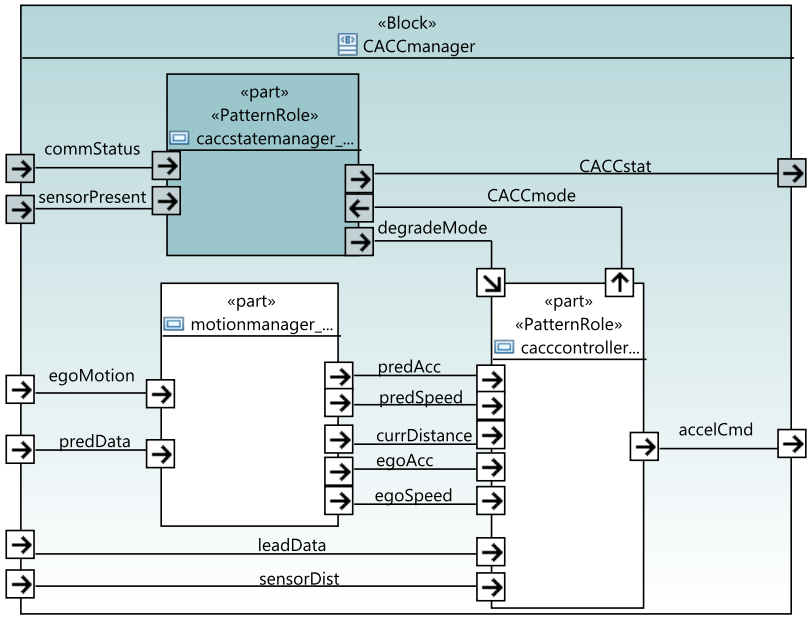

Fig. 9. The CACC manager architecture with the Monitor-Actuator design pattern

Monitor and the Actuator use the same sensory input and not redundant sensors, which means that the corresponding specifications will deal with the same variables. Finally, we compare the pattern guarantees with the related safety requirements to ensure that the pattern guarantees meet the relevant requirements. In this regard we would ensure that the integrity and availability requirements meet the demands imposed by the criticality of the hazard as mandated by the corresponding safety standard.

Security Gateway assurance: For the security gateway pattern to be applicable, we need to ensure that the communication between the vehicles is encrypted and that the gateways in all vehicles are compatible. While the security gateway adds the security layer without significantly affecting the system architecture, it does affect system timing behaviour by increasing the latency of end-to-end communication between the different CACC controllers. As long as that latency does not violate the timing safety requirements, the security gateway pattern consequences are acceptable within the context of the CACC system. Arguing further the rationale behind applying this design pattern, we match both the pattern problem to the system problem we needed to resolve, as well as the description of the pattern solution with the corresponding high-level requirements. The implementation branch is the place where we argue that for example the implementation of the gateway fulfills the message integrity demands (e.g., that

the gateway does not change the contents of the message sent and received) as well as the timing demands that should not be broken for the system to meet its timing requirements. Finally, the pattern guarantees that the gateway will reject or grant messages based on their security layer matches the corresponding requirement FSR2 as stated in Table 1.

Based on the specified design template, and the defined requirements and specification in the CHESS model, we use CHESS to automatically instantiate the argument-fragment for architectural pattern contract-based assurance presented in Fig. 2. An example of the generated argument-fragment for the application of the security pattern in our case study is shown in Fig. 10. Due to space constraints, the node statements in the argument in Fig. 10 got shortened so the three dots were used in places where the entire statement could not be previewed. A similar argument is generated for each specified design pattern application in the CHESS model. The example argument shows how the data from the design pattern template is used to populate the argument regarding the design pattern application. Developing goals related to the implementation are out of scope of this paper.

4.5 Discussion

Extending design pattern templates with contractual specifications aims at enabling clear understanding of what an environment needs to fulfil in order for the design pattern to provide its guaranteed behaviours. Conceptually, such contracts can be specified using a range of representations from natural text to formal languages. However, in practice, it is not easy to formally specify all the requirements that a system should meet for a pattern to be applied successfully. In our methodology, we envisage the possibility of specifying design pattern contracts formally such that automatic checking of design pattern compatibility is possible, but in the implementation in CHESS we have only provided support for informal contracts. In this case study, we find that even just having informal design pattern contracts is useful during the design pattern application, although greater benefits can be reaped if those contracts could be captured formally. CHESS does support formal specification and checking of contracts using Othello Specification language, used by the OCRA tool [18] for system property verification. However, further investigation is needed to determine the expressiveness of such specification language for design pattern contracts.

A general concern when using assume-guarantee style contracts lies in their completeness, i.e., have we identified all of the assumptions and have we defined all of the consequences in form of guarantees. It is impossible to prove that a component will always provide the given guarantees in any environment that meets its assumptions. Instead, contracts can be proved complete in that sense only in a subset of all environments constrained by the variables used to define the contract. Hence, contracts are said to be inherently incomplete [19]. One way to tackle their inherent incompleteness is by using confidence arguments about the contract completeness. In terms of completeness of design pattern contracts, evidence that could support it includes previous usages of the design pattern

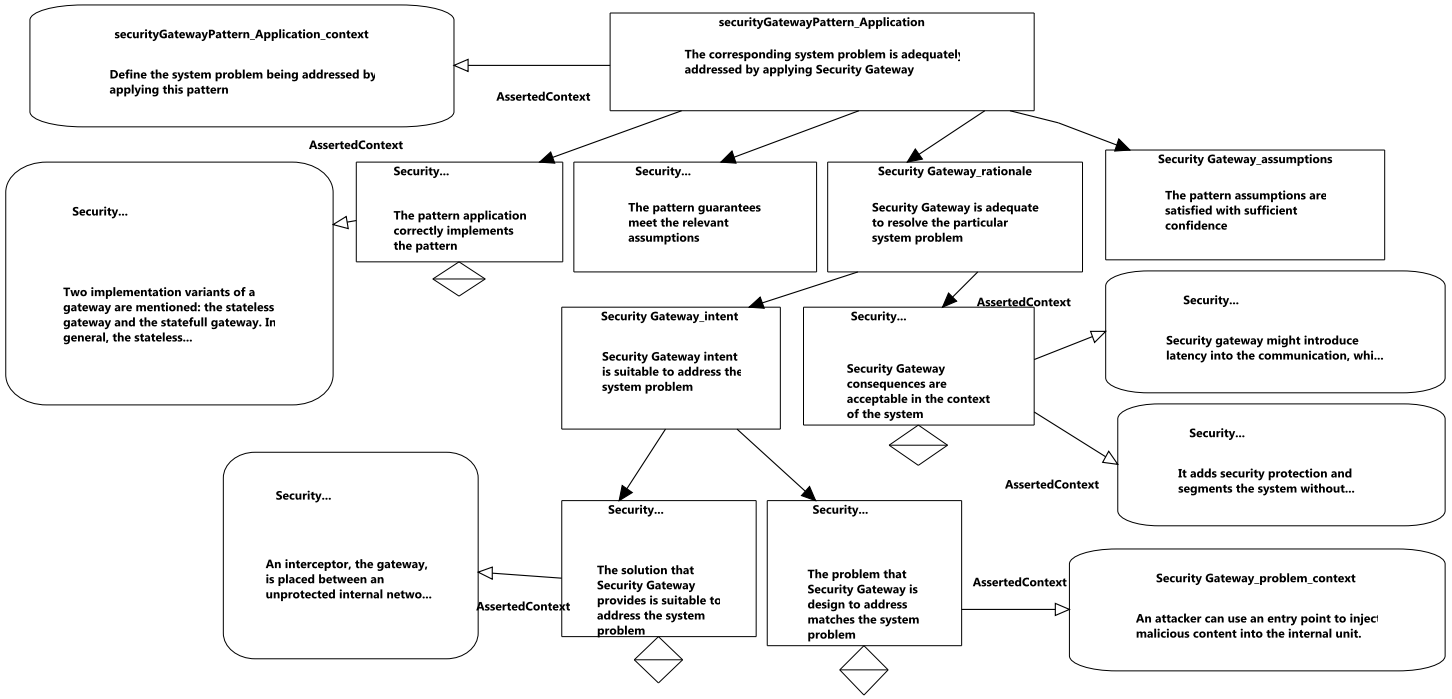

Fig. 10. An example of the automatically generated argument in OpenCert for the security gateway design pattern

and the similarity of the context in which it was used before to the context in which it will be applied. The argument pattern for design pattern application assurance that we propose in this paper focuses on the design pattern template information and the way it is implemented. The confidence argument regarding the completeness of the design pattern contract relates to whether the current application of the design pattern in terms of its implementation in the current context behaves accordingly (the confidence goal in Fig. 2). However, since we did not focus on the implementation in this case study, we did not fully develop that goal in the assurance argument in Fig. 10.

5 Related Work

Design patterns were proposed by Alexander [1] in order to establish a common solution to recurring design problems. Design patterns help designers and system architects when choosing suitable solutions for commonly recurring design problems. The use of patterns speeds architecture specification and facilitates the (re)use of components, targeted at being used in such patterns, as well as analysis results associated with the patterns.

Along the years several design patterns have been proposed tackling different concerns in Cyber-Physical Systems design such as safety (especially, via fault tolerance means) and security. Some of the most remarkable design patterns are the ones adopted in the design of safety-critical systems presented in form of a catalogue by Douglas [20], [21].

Starting from those patterns Armoush [7] proposed a template for defining design patterns template. However, his template offers no support for arguments generation and corresponding facilitation of the assurance and certification processes. In this direction, Gleirscher and Kugele summarized the applied research on design and argument patterns for the assurance of system safety [22]. Their work extended the study started in [23] to a larger context. We complement their study taking into account not only safety but security properties.

Concerning security patterns, both academic researchers and industry have contributed to their definition [24]. Although security patterns have been investigated, their role, when designing safety-critical systems, has not been sufficiently considered. Amorim et al. [17] proposed a systematic pattern-based approach that interlinks safety and security patterns and provides guidance with respect to selection and combination of both types of patterns in the context of system engineering. They developed the so-called Pattern Engineering Lifecycle, which provides a systematic way of safety- and security-related pattern engineering process. The feasibility of their approach was applied to an automotive use case.

Other domains such as industrial automation have already described security patterns as part of their certification framework. In detail, the industrial automation and control systems (IACS) Cybersecurity Certification Framework defines a protection profile of an industrial firewall in the industrial automation domain [25]. This industrial firewall can be used to segregate networks of different criticalities and to protect Industrial Control Systems. Moreover, the

documented security requirements can be translated as a set of assumptions (*ToE adnministrators are competent, trained and trustworthy*) and guarantees (*The ToE guarantees also non replay of exchanges.*) by means of the solution presented in this work.

Similarly, Preschern et al. [26] evaluated existing safety patterns regarding their effect on the overall system security. The security analysis of safety patterns was carried out by using the STRIDE approach (which stands for spoofing, tampering, repudiation, information disclosure, and escalation of privileges) and the highly critical threats for each pattern presented in a Goal Structuring Notation diagram. This work was further elaborated in [27], where GSN diagrams were related to the patterns to provide a structured overview of their architectural decisions. They presented a system of safety patterns and describe their relationships to each other.

Hamid and Perez proposed a pattern-based development approach to address dependability through a model-driven engineering approach [28]. The approach is composed of several steps and consists of metamodelling techniques that enable the specification of dependability patterns. Besides, Radermacher et al. [29] combined design patterns and model driven engineering techniques for building component-based applications with safety requirements.

Object Management Group (OMG) has published The Structured Patterns Metamodel Standard (SPMS) [30] as an industry standard that defines the definition and description of patterns used in architecting, designing, and implementing software systems. The definition of design patterns in CHESS is partially compliant to this standardised metamodel, in particular with respect to the definition of patterns. While a pattern definition is represented in terms of Roles and Sections in SPSM, CHESS uses only Roles to define a pattern, but not the sections. In addition to SPSM, we extend the pattern definition with assume/guarantee contracts, which is reflected both in the proposed methodology, and its CHESS implementation.

In comparison to our approach, none of the aforementioned works uses both multi-concern (safety and security) contracts for architectural patterns and relates them with their corresponding assurance arguments in order to facilitate the assurance and certification processes. Table 2 highlights the differences between our methodology and pre-existing related work. Specifically, "Y" indicates that the functionality is supported, "N" indicates that the functionality is not supported, "-" indicates that nothing was mentioned.

6 Conclusions and Future Work

In this paper, we have presented a methodology for assuring the application of architectural design patterns to those critical systems (i.e., safety or security-critical) for which assurance and its documentation via assurance cases is mandatory. The basis for assuring the application of a design pattern lies in the template that defines the pattern. Since, however, this basis is not sufficient for retrieving fully structured information for assurance, we proposed to extend the

Table 2. Comparative analysis summary

Functionality/Related Work	**AMASS**	**Armoush [7]**	**Gleirscher et al. [22]**	**Luo [23]**	**Preschern et al. [26]**	**Hamid [28]**	**Amorim [17]**	**IACS [25]**	**SPMS [30]**
Argumentation-oriented Pattern template definition	Y	N	N	N	N	N	N	N	N
Consideration of assumption and guarantee properties in design patterns	Y	-	-	N	N	-	-	N	N
Pattern(s) selection	Y	-	-	N	N	-	-	N	N
Argumentation-oriented Multi-concern Pattern compatibility check	Y	N	N	N	N	N	N	N	N
Pattern application	Y	Y	Y	N	N	Y	N	N	Y
Pattern-based Argument fragment generation	Y	N	N	N	N	N	N	N	N

templates with assumption/guarantee contracts. We used the extended template to guide the assurance of the application of the corresponding design pattern by proposing a safety case argument pattern. The proposed argumentation pattern identifies the evidence that should be gathered for the design pattern application assurance. We have shown that the tool-support through the AMASS platform can be used to extract information from the design pattern application and automatically instantiate the proposed argumentation patterns for each design pattern application. We have used both safety and security related design patterns in our automotive case study to show that the methodology is not specific to a particular system concern such as safety, but it can be applied to different concerns.

As future work, we plan to extend our methodology and further integrate its extensions within the AMASS platform. In particular, while we have provided tool-support for specification of informal design pattern contracts, as future work, we would like to provide support for (semi-)formal design pattern contract specification. This would allow for: a finer-grained specification of the design pattern contracts, including their variability; as well as automatic recognition of possible conflicts among selected patterns and refinement checking between the pattern contracts and the contracts of their implementations. Given that the argument patterns library is currently hardcoded within the AMASS platform, we plan to implement the argument patterns library where new patterns could

be added and the user could choose from such a library which argument pattern to automatically instantiate.

Acknowledgements

This work is supported by the EU and VINNOVA via the ECSEL Joint Undertaking project AMASS (No 692474) [8].

References

[1] Alexander, C.: A Pattern Language: towns, buildings, construction. Number 2 in Center for Environmental Structure series. Oxford University Press, New York (1977)

[2] Beck, K., Johnson, R.: Patterns generate architectures. In Tokoro, M., Pareschi, R., eds.: Proceedings ECOOP '94. Volume 821 of LNCS., Bologna, Italy, Springer-Verlag (July 1994) 139–149

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

[4] Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.B., Reinkemeier, P., Sangiovanni-Vincentelli, A., Damm, W., Henzinger, T., Larsen, K.G.: Contracts for System Design. Research Report RR-8147, Inria (November 2012)

[5] Sljivo, I., Gallina, B., Carlson, J., Hansson, H., Puri, S.: A method to generate reusable safety case argument-fragments from compositional safety analysis. Journal of Systems and Software **131** (2017) 570 – 590

[6] Assurance Case Working Group of The Safety-Critical Systems Club: GSN Community Standard Version 2. Technical report (January 2018)

[7] Armoush, A., Salewski, F., Kowalewski, S.: Design pattern representation for safety-critical embedded systems. JSEA **2**(1) (2009) 1–12

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

[9] J.L. de la Vara, A. Ruiz, B. Gallina, G. Blondelle, E. Alaña, J. Herrero, F. Warg, M. Skoglund: The AMASS Approach for Assurance and Certification of Critical Systems. In: embedded world Conference (ewC), Nuremberg, Germany. (February 2019)

[10] Javed, M.A., Gallina, B.: Get EPF Composer back to the future: A trip from Galileo to Photon after 11 years. In: EclipseCon, Toulouse, France, June 13-14. (2018)

[11] Soundarajan, N., Hallstrom, J.O.: Responsibilities and rewards: Specifying design patterns. In Finkelstein, A., Estublier, J., Rosenblum, D.S., eds.: 26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom, IEEE Computer Society (2004) 666–675

[12] International Organization for Standardization: ISO/IEC 15026: Systems and software engineering – Systems and software assurance. (2011)

[13] Object Management Group: UML: Unified Modeling Language. Technical report, Version 2.5.1, OMG, 2017. https://www.omg.org/spec/UML/2.5.1/

[14] Sljivo, I., Gallina, B., Kaiser, B.: Assuring degradation cascades of car platoons via contracts. In Stefano Tonetta, Erwin Schoitsch, F.B., ed.: 6th International Workshop on Next Generation of System Assurance Approaches for Safety-Critical Systems. Volume 10489., Springer (September 2017) 317–329
[15] International Organization for Standardization: ISO 26262: Road vehicles — Functional safety. (2011)
[16] Armoush, A.: Design Patterns for Safety-CriticalEmbedded Systems. PhD thesis, Aachen University (June 2010)
[17] Amorim, T., Martin, H., Ma, Z., Schmittner, C., Schneider, D., Macher, G., Winkler, B., Krammer, M., Kreiner, C.: Systematic pattern approach for safety and security co-engineering in the automotive domain. In: Computer Safety, Reliability, and Security, Cham, Springer International Publishing (2017) 329–342
[18] Cimatti, A., Tonetta, S.: Contracts-refinement proof system for component-based embedded systems. Science of Computer Programming **97**(3) (2014) 333–348
[19] Sljivo, I., Jaradat, O., Bate, I., Graydon, P.: Deriving safety contracts to support architecture design of safety critical systems. In: 16th IEEE International Symposium on High Assurance Systems Engineering, IEEE (January 2015) 126–133
[20] Douglass, B.P.: Doing hard time: Developing real-time system with UML, objects, frameworks, and pattern. New York: Addison-Wesley (1999)
[21] Douglass, B.P.: Agile Systems Engineering. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2016)
[22] Gleirscher, M., Kugele, S.: Assurance of system safety: A survey of design and argument patterns. CoRR **abs/1902.05537** (2019)
[23] Y. Luo, Z., van den Brand, M.: A categorisation of gsn-based safety cases and patterns. In: 2016 4th International Conference on Model-driven Engineering and Software Development(MODELSWARD). (2016) 509–516
[24] Schumacher, M.: Security Engineering with Patterns: Origins, Theoretical Models, and New Applications. Springer, Heidelberg (2003)
[25] IACS: IACS Cybersecurity Certification Framework. Technical report, IACS (July 2018)
[26] Preschern, C., Kajtazovic, N., Kreiner, C.: Security analysis of safety patterns. In: Proceedings of the 20th Conference on Pattern Languages of Programs, The Hillside Group (2013)
[27] Preschern, C., Kajtazovic, N., Kreiner, C.: Building a safety architecture pattern system. In: Proceedings of the 18th European Conference on Pattern Languages of Program. EuroPLoP '13, New York, NY, USA, ACM (2015) 17:1–17:55
[28] Hamid, B., Perez, J.: Supporting Pattern-Based Dependability Engineering via Model-Driven Development: Approach, tool-support and empirical validation. Journal of Systems and Software **vol. 122** (September 2016) pp. 239–273
[29] Radermacher, A., Hamid, B., Fredj, M., Profizi, J.L.: Process and tool support for design patterns with safety requirements. In: Proceedings of the 18th European Conference on Pattern Languages of Program. EuroPLoP '13, New York, NY, USA, ACM (2015) 8:1–8:16
[30] Object Management Group: SPMS: Structured Patterns Metamodel Standard. Technical report, Version 1.2, OMG, 2017. https://www.omg.org/spec/SPMS/