The Journal of Systems & Software 176 (2021) 110922

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Product-line assurance cases from contract-based design ☆

Damir Nešić [a,*], Mattias Nyberg [a], Barbara Gallina [b]

[a] KTH Royal Institute of Technology, Brinellvägen 83, 100 44 Stockholm, Sweden
[b] Mälardalen University, Högskoleplan 1, 722 20 Västerås, Sweden

ARTICLE INFO

Article history:
Received 25 November 2019
Received in revised form 6 November 2020
Accepted 2 February 2021
Available online 10 February 2021

Keywords:
Assurance cases
Product line engineering
Contract-based design

ABSTRACT

Assurance cases are used to argue in a structured, and evidence-supported way, that a property such as safety or security is satisfied by a system. In some domains however, instead of single systems, product lines with many system-variants are engineered, to satisfy the needs of different customers. In such context, single-system methods for assurance-case creation suffer from scalability issues because the underlying assumption is that the evidence and arguments can be created per system variant. This paper presents a novel method for product-line assurance-case creation where all the arguments and the evidence are created without analyzing each system variant. Consequently, the effort to create an assurance case scales with the complexity of system variants, instead with their number. The method is based on a contract-based design framework for cyber–physical systems, which is extended to define the conditions under which all system variants satisfy a particular property. These conditions are used to define an assurance-case pattern, which can be instantiated for arbitrary product lines. Moreover, the defined pattern is modular to enable step-wise assurance-case creation. Finally, an exploratory case study is performed on a real product-line from the heavy-vehicle manufacturer Scania to evaluate the applicability of the presented method.

1. Introduction

An assurance case is *"a documented body of evidence that provides a convincing and valid argument that a specified set of critical claims regarding a system's properties are adequately justified for a given application in a given environment"* (Rhodes et al., 2010). Similar to a court case, an assurance case is a *structured*, *evidence-based* way to present the arguments that a particular system is acceptably safe, acceptably secure etc. Conceptually, an assurance case contains two types of arguments, product-based and process-based. The former should show that the system behaves in accordance with its specification, while the latter should show that the engineering personnel, processes, and tools cannot accidentally introduce flaws into the system. Currently, assurance cases are recommended or even required by various standards, e.g. ISO 26262 (International Organization for Standardization, 2011) or EN 50129 (European Committee for Electrotechnical Standardization, 2018), including the recent, first-ever standard for the safety of autonomous systems, UL 4600 (UL4600 Task Group, 2020).

To support different operational environments, and satisfy the needs of different markets and corresponding legislative frameworks, companies typically develop *families of similar systems*, called *Product Lines* (Linden et al., 2007; Clements and Northrop, 2001), and not *single systems*. Examples of product lines are everywhere, from open-source software product-lines such as the Linux and eCos operating systems (Hubaux et al., 2012), to product lines in critical domains such as industrial automation (Fogdal et al., 2016), automotive (Wozniak and Clements, 2015), aerospace (Gaeta and Czarnecki, 2015), railway (Svendsen et al., 2010) etc. Independently of the domain, product lines are typically designed as *integrated platforms* from which the members of the family, called *variants*, can be derived. Consequently, when creating assurance cases, companies are faced with the challenge of arguing that *all variants* are acceptably safe, secure etc., instead of the classical case with a single system.

Product lines come with two fundamental challenges that hinder the creation of corresponding assurance cases. Firstly, a significant number of product lines define thousands and even millions of possible system variants (Mukelabai et al., 2018; Metzger and Pohl, 2014; Wozniak and Clements, 2015; Dietrich et al., 2012). Consequently, *exhaustive verification* of *all derivable system variants* is typically not practised because of the high cost. For this reason, *verification evidence* is not produced for each system variant and creating an assurance case per system variant becomes unfeasible.

☆ Editor: Raffaela Mirandola.

* Corresponding author.
E-mail addresses: damirn@kth.se (D. Nešić), matny@kth.se (M. Nyberg), barbara.gallina@mdh.se (B. Gallina).

https://doi.org/10.1016/j.jss.2021.110922
0164-1212/

The second challenge relates to the analysis of product-line assets. Typically, engineering assets of a product line are created so that they apply to multiple system variants. For example, a piece of source code could belong to many variants, but each variant will execute only a part of the source code. The consequence is that the classical, *single-system* analysis methods can only be used if an asset is reduced to a version that belongs to a single variant. To avoid this overhead, different analysis methods are modified to become *variability-aware* (Thüm et al., 2014), i.e. they are designed to analyze an asset *as-is* by considering the *built-in variability*. Preferably, the same principles should exist for an assurance case that argues about multiple variants, i.e. there should exist a set of *variability-aware* conditions that define if an assurance case is *complete, correct etc.*

1.1. Main related work

This section reviews existing methods that tackle the problem of creating an assurance case for a product line, to highlight their limitations. Methods that deal with parts of this problem, e.g. how to represent the relevant artifacts of a product line, or how to create assurance-case arguments that can be reused in different contexts, are considered in Section 9.

There are two notable lines of research for systematic creation of assurance cases for product lines where both focus on per-variant *assurance-case* creation, with primarily *product-based* arguments about safety properties.

The first line of research (Habli, 2009; Habli and Kelly, 2010; de Oliveira et al., 2015) presents a generic, product-line safety meta-model that defines generic traceability links and generic assets, which can be *common* or *variable*. Examples of traceability links are *associated with*, or *generated from*, while *safety analysis assets* or *assurance-case assets* are examples of generic assets. The meta-model can then be used to structure concrete assets in a concrete domain to support traceable product-line engineering process. Furthermore, the *Goal-Structuring Notation* (GSN) (Origin Consulting (York) Limited, 2018) is used to express assurance-case arguments, which reflect the variability within the product-line assets. In other words, the method yields an *integrated* product-line assurance-case argument that argues about multiple system variants, in the same way as product-line assets belong to multiple system variants. Then, just as system variants are derived from the product line, per variant assurance-case can be derived from the integrated product-line assurance case.

While the work in Habli (2009), Habli and Kelly (2010) and de Oliveira et al. (2015) supports more systematic product-line safety analyses, and the creation of an integrated product-line assurance case, the two fundamental challenges in product line engineering are not addressed. Firstly, the method *assumes that per-variant assurance cases will be derived*, and analyzed for completeness and consistency, i.e. there are no *variability-aware* criteria to analyze the assurance case. This means that the integrated assurance case is essentially a superimposition of multiple per-variant assurance cases. Secondly, as a consequence of the first issue, the verification *evidence must be created per system-variant* which, as discussed previously, is typically not feasible.

The second line of research (Hutchesson and McDermid, 2010, 2011, 2013) focuses on the aerospace domain, and considers pure software assurance. The method relies on a *model-to-model transformation* from a UML class-model of a software system, to SPARK Ada programs (Chapman and Schanda, 2014). The UML class-models contain parametrized annotations, which define the properties that different system variants must satisfy. During model-to-model transformation, which corresponds to variant derivation, all parametrized annotations are resolved, and a particular system variant is derived. Then, off-the-shelf tools can be used to automatically analyze per-variant SPARK Ada programs in order to guarantee data-flow properties, functional correctness etc. In other words, the method allows producing evidence which supports product-based arguments that a software variant satisfies a certain property. As an extension, the work in (Hutchesson and McDermid, 2013) discusses the notion of *trustworthiness*, and defines several process-based arguments, which are required to claim that each variant *derived* from a product line is safe.

The notion of trustworthiness in (Hutchesson and McDermid, 2010, 2011, 2013), coupled with SPARK Ada definitions of a component satisfying a property, explicitly shows why the resulting arguments are *complete and correct*. However, although automatically, both the verification evidence and the analysis for completeness and correctness is still performed on a per variant basis. Furthermore, because properties such as safety or security are *system-level properties*, the relation between the claim that a software component will operate as intended, and the claim that the overall system containing the software will operate as intended, is undefined.

1.2. Paper contribution

Given the challenges of product line engineering, and the limitations of previous method for assurance-case creation for product lines, the goal of the present paper is to define a novel and *general* method for the creation of assurance-case argumentation for *all variants* that can be derived from a product line, *without* actually performing the derivation.

To this end, the current paper presents a method based on a formal *Contract-Based Design* (CBD) framework (Westman and Nyberg, 2018a; Benveniste et al., 2018; Cimatti and Tonetta, 2012). Because existing CBD frameworks were developed for *single system analysis*, through a series of formal definitions, theorems, and deductive proofs, the first contribution of the paper is a set of *variability-aware* conditions over an extended CBD framework, which preserve the compositional nature of the original CBD framework, but allow modeling and analysis of product lines. If satisfied, these conditions allow inferring that *all possible system variants* satisfy the declared system property. Furthermore, the combination of the *variability-aware* conditions, and the compositional nature of the CBD framework, means that instead of verifying a high number of variants, it is sufficient to verify only the components that comprise the different variants.

The second contribution is the definition of two *assurance-case patterns* (Kelly and McDermid, 1997) in the GSN format, which given a CBD model of a product line, create the corresponding *product-based*, *assurance-case argumentation*. More precisely, the *assurance-case patterns* are based on the variability-aware conditions of the extended CBD framework. In this way, the argumentation obtained by instantiating the pattern argues *directly* about all possible variants of a system. Because for real systems, the assurance-case argumentation can be quite large, the presented patterns are *modular*. This supports both the separation of concerns, e.g. different engineering groups can create smaller parts of the assurance case, and also a *stepwise* creation of the assurance case in parallel with the engineering process.

To verify if the extended CBD framework can be used to model and analyze real product lines, the paper presents an *exploratory case study* (Runeson et al., 2012) by applying the framework to a real, configurable system from the heavy vehicle manufacturer Scania. The case study reveals that the CBD framework *can indeed* be applied to a real system, but depending on the available artifacts and used engineering practices, some information must be inferred by jointly analyzing the conditions of the CBD framework and the available artifacts. To evaluate the assurance-case patterns, the analyzed Scania system is used to instantiate the

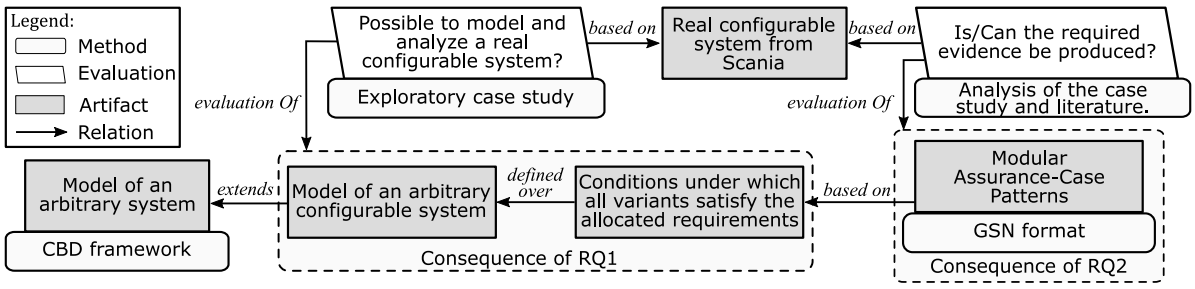


Fig. 1. Underlying methodology and relations between the elements of the present paper.

pattern. Despite the need for further validation, initial results show that the proposed modular architecture aligns rather well with the analyzed engineering process, and with the process of the functional safety standard from the automotive domain, i.e. ISO 26262. Regarding the types of evidence required by the pattern, the majority was already present among the SCANIA artifacts. However, two types were not because producing them requires more advanced, yet mature, automated methods.

The present paper extends the work in (Nešić et al., 2019) by considering a more expressive contract-based design-framework in order to support a greater array of engineering scenarios. Consequently, the product-line extension of the contract-based framework is also refined. Furthermore, and unlike (Nešić et al., 2019), an exploratory case study is performed on a part of a real product-line from the heavy vehicle manufacturer SCANIA, to evaluate the applicability of the extended contract-based framework. Finally, the assurance-case pattern is refined in order to match the refined extension of the contract-based framework, and the pattern is redefined to yield a modular assurance case.

1.3. Paper structure

Section 6.1 presents the underlying methodology of the present paper. Section 3 summarizes the basics of product lines and an existing contract-based framework. Section 4 presents minor modifications of the existing framework in order to allow explicit modeling of *assumptions* that can later be considered in the product line extension. Section 5 presents the product-line extension of the general contract-based framework, which allows modeling of configurable systems. Section 6 presents an exploratory case study that evaluates the applicability of the CBD framework to a configurable system from SCANIA. Section 7 recalls the basics of GSN format and then defines the modular assurance-case pattern. This is followed by a pattern-instantiation procedure and an example instantiation for the analyzed SCANIA system. Section 8 answers the two research question. Section 9 discusses the related work while Section 10 reviews the threats to validity for the presented method. Finally, Section 11 concludes the paper.

2. Methodology

This section discusses the methodological foundation of the present paper with the help of Fig. 1. As outlined in Section 1, the goal of the present paper is to define a *general* method for the creation of assurance-case argumentation for *all variants* that can be derived from a product line, *without* actually performing the derivation. Similarly to the methods that produce *per-variant* assurance-case argumentation, this entails creating a model of a product line and its artifacts, and a subsequent definition of a method that leverages the model to create assurance-case argumentation.

As Fig. 1 shows, the present paper relies on a CBD framework (Westman and Nyberg, 2018a; Benveniste et al., 2018; Cimatti and Tonetta, 2012) to model an arbitrary system, i.e. a configurable system. There are multiple reasons for this choice. The most important reason is the support for *compositional reasoning* (De Roever et al., 2003) and *compositional verification* (Bensalem et al., 2008), where the properties of a system can be *inferred* from the *verified properties* of its *constituent components*. Consequently, the properties of individual variants can be inferred from the properties of their constituent components, *without actually deriving individual variants*. Another important reason is that CBD frameworks formalize the typical engineering process *as-is* (Benveniste et al., 2018), i.e. they formalize in an abstract way the *vertical* design *refinement* during the development process, and *horizontal composition* of logical or physical components at a given abstraction level. However, CBD frameworks are *not intended* to be a concrete engineering framework, but rather an *orthogonal representation* of engineering artifacts, which enables *advanced, and preferably automated analyses* regardless of the used engineering practices. This is achieved by abstracting away the exact content of the artifacts and defining refinement and composition independently of the concrete syntax used to express the different artifacts. Yet another reason is that contract thinking is common in everyday engineering, and in general. First explicit contract-driven method for software development dates back to early '90s (Meyer, 1992). Furthermore, ten years ago, ISO 26262 introduced the concept of a *safety element out-of-context*, which is developed with respect to a contract and it guarantees certain functionality only when it is integrated with other systems that satisfy certain assumptions.

Despite the benefits, none of the existing CBD frameworks support modeling of configurable systems (see survey in (Benveniste et al., 2018)), i.e. product lines, thus it was necessary to extend an existing CBD framework and this has led to the first research question:

RQ1 Can a CBD framework be used to model an arbitrary product line, and support an analysis of whether all variants satisfy a particular property without performing the analysis per-variant?

As shown in Fig. 1, a consequence of RQ1 is the first contribution of the paper, namely the *extension* of an existing CBD framework with support to model and analyze configurable systems. To *evaluate* if that this is indeed *possible*, an *exploratory case study* (Runeson et al., 2012) was performed on a *real, configurable systems* from the heavy-vehicle manufacturer SCANIA.

Given the extended CBD framework, and to reach the overall goal of the paper, it was necessary to define a method that leverages the extended CBD framework to construct assurance-case argumentation. This has led to the formulation of the second research question, namely:

RQ2 How can a model of a product line in the extended CBD framework be used to construct an assurance case for an arbitrary product line?

As illustrated in Fig. 1, the developed method was encoded as several *modular assurance-case patterns* (Kelly and McDermid, 1997) expressed in GSN format. The GSN format was chosen because of an abundance of available tool-support, and because it is frequently used in academic publications. However, the encoded argumentation is in no way dependent on the format in which it is expressed. Because the modular assurance-case patterns are *based on* the conditions of the extended CBD framework, and because the expected assurance-case evidence corresponds to the *successful verification results* of these conditions, the proposed patterns are *evaluated* from the perspective of whether these conditions are practically verifiable. The analysis was done both against the identified artifacts and observed practices in SCANIA, but also against existing tool-supported methods that can verify such conditions.

3. Preliminaries

This section summarizes the basic concepts of product lines and CBD frameworks from previous literature.

3.1. Product line engineering

Product line engineering (Linden et al., 2007; Apel et al., 2016), facilitates the development of a family of systems that are jointly referred to as a *product line*. Instead of developing each system individually, product line engineering promotes the creation of an *integrated platform* that *contains* the artifacts of *all variants*, and then individual variants are *derived* from the integrated platform. To distinguish between variants on a more abstract level than the corresponding artifacts, product line engineering promotes the use of *features*. Features are abstractions of functional and non-functional characteristics of each system variant, that are understandable for the customers of the system. For example, *automatic* and *manual transmission* could be *features* within an automotive product-line. Features, and their dependencies, are captured through *variability models*.

A product line can be implemented in many ways (Dubinsky et al., 2013; Liebig et al., 2010) but there is a wide agreement that independently of the selected technologies, the mechanisms underlying product line implementation is either *annotative* or *compositional* (Kästner and Apel, 2008). In annotation-based product lines, each artifact is *annotated* with the *features to which it applies*. By *selecting* a set of features, the corresponding artifacts are indirectly selected, and a system variant is derived. In compositional product lines, the implementation of each feature is a single *'unit'*, e.g. a particular software class, and when a set of features is selected, the corresponding units are *composed* into a system variant.

Compositional approaches are suitable for software product lines where a compiler guarantees the *correct composition* of source-code units. However, composition of other artifacts is not always defined, and rarely tool-supported, e.g. composition of requirements. Thus, product lines are typically implemented using the *annotative* mechanism, which is the one considered in the remainder of the paper.

3.1.1. Annotation-based product lines

Let $\mathcal{F}$ be a set of *features* of a *product line* where each feature $f_i \in \mathcal{F}$ is a Boolean variable. Features can also be of other types, e.g. numerical or enumerations, but this is a trivial extension and is not considered further to avoid notational overhead.

Definition 1 (*Variability Model*)**.** A *variability model* is a pair $\mathfrak{M} = (\mathcal{F}, \Theta)$ where $\mathcal{F}$ is a set of features and Θ is a set of Boolean formulas over the features in $\mathcal{F}$. □

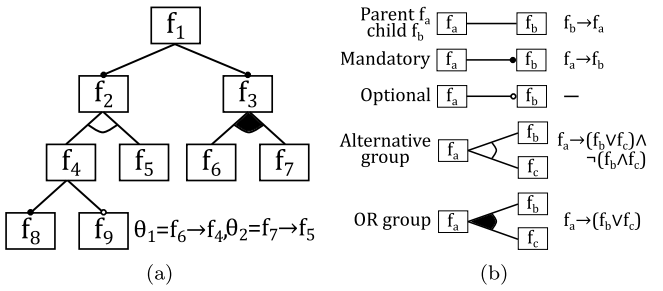


Fig. 2. Example feature model (a), and its encoding as Boolean formulas (b).

There are multiple types of variability models introduced in the literature (Czarnecki et al., 2012), but the *de-facto* standard is a *feature model* (Nešić et al., 2019). The idea behind a feature model, as shown in Fig. 2(a), is to visualize the features, and a subset of the Boolean formulas as a tree-structure that is intuitive to build, understand, and maintain. Within a feature model, features can be declared to be *mandatory*, *optional* etc., and Fig. 2(b) shows how the different feature-model constructs are encoded as Boolean formulas within Θ. For example, features f_4 and f_5 could represent the *automatic* and *manual* transmission, that are mutually exclusive, while their parent feature f_2 could represent *transmission* feature that is *mandatory*. As Fig. 2(a) shows, additional Boolean formulas outside the tree-structure are maintained in a textual format, e.g. θ_i and θ_j in Fig. 2(a). Variability models allow the selection of *configurations*.

Definition 2 (*Configuration*)**.** Given a variability model $\mathfrak{M}$, a *configuration* γ is a set of feature-value assignments $\gamma = \{f_i = value_k\}$ for each $f_i \in \mathcal{F}$. A configuration for which each formula in Θ evaluates to true is *valid*, and the set of valid configurations is denoted Γ. □

The previously informal notion of *selecting a feature* corresponds to assigning the value *true* to a feature, while *de-selecting* corresponds to assigning the value *false*. If a value is assigned to each of the features from $\mathcal{F}$ then a *configuration is selected*. To highlight an *invalid* configuration, consider a γ for the feature model in Fig. 2 where $\{f_2 = true, f_4 = true, f_5 = true\} \subseteq \gamma$. For such γ, the constraint for the alternative group f_2, f_4, f_5 would evaluate to *false*, thus $\gamma \notin \Gamma$. On an intuitive level, if f_4 and f_5 represent automatic and manual transmission, then the feature model in Fig. 2 declares that a vehicle *cannot simultaneously* have both types of transmission. Hereinafter, we will consider only the set of *valid configurations* Γ. Variability models of real-world product lines often contain *thousands of features*, and consequently define *billions of valid configurations* (Mukelabai et al., 2018).

Given a variability model, the artifacts of the integrated platform are *annotated* with *presence conditions* (Rhein et al., 2015).

Definition 3 (*Presence Condition*)**.** A *presence condition*, denoted φ, is a Boolean formula over the features from $\mathcal{F}$ from a variability model $\mathfrak{M}$, conforming to the grammar $\varphi ::= f_i | \neg\varphi | \varphi \wedge \varphi | \varphi \vee \varphi$. □

The purpose of a presence condition is to define a subset of Γ, to which an artifact applies. For example, if a presence condition is $f_1 \wedge f_4$, then this presence condition evaluates to true for all configurations that contain value assignments $f_1 = true$ and $f_4 = true$. Formally, let function *eval* be such that given a presence condition φ and a configuration γ it returns a value *true* or *false*. Note that we will consider only presence conditions that are *consistent* with the given variability model $\mathfrak{M}$, i.e. for each φ, there exists a γ such that $eval(\varphi, \gamma) = true$. For example, the

feature model in Fig. 2(a) excludes the presence condition $f_9 \wedge f_{10}$ because f_{10} is not defined in the feature model.

The function *eval* models the so-called *product configurator* that allows a customer to select a *configuration* of the product in terms of features, and then evaluates the presence condition of all artifacts within the *integrated platform*. The artifacts whose presence conditions evaluate to *true*, are the ones that belong to the *variant* that corresponds to the selected configuration. To analyze the properties of each variant, e.g. safety or security, it is necessary to have a model of the artifacts of the integrated platform, e.g. logical and technical components, requirements etc.

3.2. A contract-based design framework

This section summarizes the used CBD concepts from (Westman and Nyberg, 2015; Benveniste et al., 2018), with a slightly different notation that highlights the use of CBD for requirements engineering. The axiomatic concepts of the CBD framework are:

(i) *component* C,
(ii) *specification* S,
(iii) component *satisfies* a specification, denoted as $C \triangleright S$
(iv) specification *refines* a specification, denoted $S_1 \sqsubseteq S_2$
(v) *commutative* and *associative* n-ary component *composition* that results in new components, denoted as $C = C_1 \otimes \cdots \otimes C_n$ or shortly $C = \bigotimes_{i=1}^{n} C_i$.

The purpose of the above concepts is to represent central artifacts and activities during systems engineering. Components represent physical or logical components of a system, while the *composition* operation represents the common engineering activity of integrating components into systems. A specification defines the intended behavior of a component, and the *satisfies* relation represents the observation that a component behaves as defined by a specification. Specification refinement means that if a component C *satisfies* S_1, and if S_1 *refines* S_2, then the component C *also satisfies* S_2. Depending on the interpretation of specifications, refinement can correspond to *logical entailment* between logical formulas, to the *subset relation* between the sets of possible behaviors, etc. However, being axiomatic, no particular formalism is assumed for the above concepts. For example, a specification can be a temporal-logic formula, a differential equation, or plain natural-language text.

We refer to components that are not composed of other components as *atomic components*, and to components composed of other components as *composite components*. We also say that a composite component has *subcomponents*. To capture the composition of components into larger components, i.e. a system, we introduce the concepts of an *architecture* where the terminology related to trees, and graphs in general, is in accordance with (Rosen and Krithivasan, 2012).

Definition 4 (*Architecture*)**.** An *architecture* $\mathscr{A}$ is a finite, non-empty set of components organized into a *rooted tree* where each *leaf* node is an *atomic* component, and each *non-leaf* node C with children nodes $C_1, \ldots, C_n$ is a *composite component*, such that $C = \bigotimes_{i=1}^{n} C_i$. □

Note that Definition 4 does not place any constraints on the types of components that can be composed into an architecture, e.g., SW and HW components can be composed into the same architecture. As stated above, *specifications* define the intended behavior of components. As is often the case, several specifications define the behavior of a component, and to capture this scenario we define *specification conjunction*.

Definition 5 (*Specification Conjunction*)**.** The *conjunction* of specifications S_1 and S_2, denoted $S_1 \sqcap S_2$, is a specification such that $C \triangleright (S_1 \sqcap S_2) \Leftrightarrow C \triangleright S_1 \wedge C \triangleright S_2$. □

To be able to split the responsibilities on a component and on other components in its environment, the concept of a *contract* is introduced.

Definition 6 (*Contract*)**.** A contract K is an ordered pair, denoted $(\mathcal{A}, G)$, where $\mathcal{A}$ is a possibly empty, finite set of specifications called *assumptions*, i.e. $\mathcal{A} = \{A_i\}_i$, and G is a specification called a *guarantee*. □

The contracts defined in Definition 6 are referred to as *assume-guarantee* contracts and guarantees are used to express relevant properties that a component should satisfy, e.g. a safety or a security property. As a notational convention, the indices of a contract, the corresponding set of assumptions, and the corresponding guarantee always match, e.g. $K_j = (\mathcal{A}_j, G_j)$, while a particular assumption within $\mathcal{A}_j$ is denoted A_i^j.

Some CBD frameworks consider contracts also to be specifications. Here, we distinguish between them because a specification does not necessarily correspond to the *assume-guarantee* relation specified by a contract.

The idea of separating the responsibilities between a component and its environment can be best seen from the following definition, where the notation $C \triangleright \mathcal{A}_j$ is a shorthand for $C \triangleright A_1^j \sqcap \cdots \sqcap A_n^j$ where each $A_i^j \in \mathcal{A}_j$.

Definition 7 (*Satisfy Contract*)**.** Component C *satisfies* a contract $(\mathcal{A}_j, G_j)$, denoted $C \blacktriangleright (\mathcal{A}_j, G_j)$, if

$$\forall C_e . C_e \triangleright \mathcal{A}_j \rightarrow C_e \otimes C \triangleright G_j. \quad \square$$

Less formally, given a contract $(\mathcal{A}_j, G_j)$, a component C can be developed independently of other components, with respect to the contract. Then, whenever component C is composed with a component C_e, often referred as the *environment* of C, which implements each assumption in $\mathcal{A}_j$, then the composition of C and C_e implements G_j. To express that a component C should satisfy a contract K, relation *allocated to* is introduced, denoted $\texttt{allTo}(K, C)$.

Finally, it should be noted that the presented CBD framework assumes that the specification satisfaction is *monotonic* with respect to component composition. This means that if a component C_1 satisfies a specification S, once it is composed with another component C_2, their composition still satisfies S. Although monotonicity does not hold in some cases (Westman and Nyberg, 2018b), the remainder of the paper assumes monotonicity because capturing non-monotonic cases only extends but does not conflict with any of the presented CBD concepts.

4. Specification structure

With the goal to obtain a *holistic, and syntactic model of a system*, this section introduces the *specification structure*, which contains all the contracts, their relations, and the components of a system. A similar model, called a *contract-structure*, has been presented previously (Westman and Nyberg, 2015) but it contained only the guarantees of contracts. In the following definition the notation $S_1 \equiv S_2$ means that S_1 and S_2 are the same specification, and a *graph* without *loops* is a graph without cycles of lenght one (Rosen and Krithivasan, 2012, p. 594).

Definition 8 (*Specification Structure*)**.** Let $\mathscr{A}$ be an architecture with the root component C_r. Let $\mathcal{S}$ be a finite, non-empty set of specifications that form the set of contracts $\mathcal{K}$, where each $K \in \mathcal{K}$ is allocated to at least one component from $\mathscr{A}$. Then, a *specification structure* $\mathfrak{D}$ for $\mathcal{S}$ is a tuple $\mathfrak{D} = (\mathcal{N}, \mathcal{E}, \mathscr{A})$, where

(i) each node $n \in \mathcal{N}$ represents a single specification $S \in \mathcal{S}$, and each $S \in \mathcal{S}$ is represented by a single $n \in \mathcal{N}$,
(ii) each edge $e \in \mathcal{E}$ represents a single *refines* or *assumption of* relation, denoted $e^{\mathtt{r}} = (S_1, S_2)$ and $e^{\mathtt{a}} = (S_1, S_2)$,
(iii) the graph formed by $\mathcal{N}$ and $\mathcal{E}$ *does not contain loops*,
(iv) for each $e^{\mathtt{a}} = (S_1, S_2)$, there exists a contract $(\mathcal{A}_j, G_j) \in \mathcal{K}$ such that $S_1 \in \mathcal{A}_j$ and $S_2 \equiv G_j$,
(v) for each $e^{\mathtt{r}} = (S, A_i^j)$, where A_i^j is an assumption of a contract allocated to $C_m \in \mathscr{A}$, it holds that

(a) if $S \equiv A_i^k$ of a contract $K_k \in \mathcal{K}$, then there exists a $C_p \in \mathscr{A}$ such that C_p is the parent of C_m and $\mathtt{allTo}(K_k, C_p)$,
(b) if $S \equiv G_k$ of a contract $K_k \in \mathcal{K}$, then there exist $C_p \in \mathscr{A}$ such that C_m and C_p are siblings and $\mathtt{allTo}(K_k, C_p)$.

(vi) for each $e^{\mathtt{r}} = (S, G_j)$, where G_j is a guarantee of a contract allocated to $C_m \in \mathscr{A}$, it holds that $S \equiv G_k$ of a contract $K_k \in \mathcal{K}$ and there exists a $C_p \in \mathscr{A}$ such that C_m is the parent of C_p, and $\mathtt{allTo}(K_k, C_p)$. □

To provide intuition for the conditions of Definition 8, Fig. 3(a) visualizes a toy specification structure. The corresponding architecture, shown in Fig. 3(b), contains the composite component $\mathtt{C_{sys}}$, which is composed of atomic components $\mathtt{C_1}$-$\mathtt{C_3}$. The specification structure in Fig. 3(a) is formed for specifications $\mathtt{A_1}$-$\mathtt{A_4}$ and $\mathtt{G_1}$-$\mathtt{G_4}$, which in turn form contracts $(\{\mathtt{A_1}\}, \mathtt{G_1})$, $(\emptyset, \mathtt{G_2})$, $(\{\mathtt{A_2}, \mathtt{A_3}\}, \mathtt{G_3})$, and $(\{\mathtt{A_4}, \mathtt{A_2}\}, \mathtt{G_4})$. As can be seen, each specification is a node in a graph (condition (i)), and for each contract there exists an edge labeled a, to represent the *assumption of* relation between each assumption and the corresponding guarantee (condition (ii) and (iv)). Furthermore, the remaining edges are labeled r, to represent the *intended refinement* between pairs of specifications (condition (ii)). Note that for readability reasons, the graphical representation in Fig. 3(a) contains multiple nodes per specification, e.g. two nodes labeled $\mathtt{A_2}$, but formally, Definition 8–(i) allows a single node per specification and vice-versa.

While conditions (i)–(iv) establish the correspondence between specifications, contracts, specification-relations and the graph of a specification structure, conditions (v) and (vi) are different in nature. They define constraints on the possible edges, and consequently on the relations that these edges represent. First, note that the *allocated to* relation is visualized by *overlaying* a contract over a component, e.g. in Fig. 3(a) it holds that $\mathtt{allTo}((\{\mathtt{A_1}\}, \mathtt{G_1}), \mathtt{C_{sys}})$. Condition (v) states that if a contract is allocated to a component, e.g. $\mathtt{allTo}((\{\mathtt{A_2}, \mathtt{A_3}\}, \mathtt{G_3}), \mathtt{C_2})$, then the assumptions can be refined either by a guarantee of a contract allocated to a sibling component, e.g. $\mathtt{G_2}$, or by an assumption of contract allocated to the parent component, e.g. $\mathtt{A_1}$. Conceptually speaking, if a component assumes some functionality, then this functionality is either implemented by a component of the same system (condition (v)–(b)), or the whole system assumes this functionality from some other system (condition (v)–(a)). Condition (vi) states that if a contract is allocated to a composite component, e.g. $\mathtt{allTo}((\{\mathtt{A_1}\}, \mathtt{G_1}), \mathtt{C_{sys}})$, then the guarantee of that contract can be refined *only* by a guarantee of a contract allocated to a subcomponent of the composite component, e.g. $\mathtt{G_4}$. Conceptually, this means that any functionality implemented by a composite component, e.g. a system, must be a consequence of the functionality implemented by some of its constituent components.

Note that within a particular company, artifacts such as different types of components and specifications reside in different tools, and are expressed in multiple notations. The purpose of a specification structure is to be an *integrated representation* of all these artifacts and not a *replacement*. This representation should be *constructed on-demand for analysis purposes*, and although possible, it is not intended to be used as a concrete modeling-framework in everyday engineering. When it comes to the possible analyses, for example, the specification structure matches well with the principles from several functional safety standards (International Organization for Standardization, 2011; The International Electrotechnical Commission, 2010). Namely, they require a decomposition of higher level requirements into lower level requirements, e.g. refinement of $\mathtt{G_1}$ into $\mathtt{G_4}$, in conjunction with the decomposition of the system architecture, e.g. $\mathtt{C_{sys}}$ is composed of $\mathtt{C_3}$. The existence of such formal, integrated model of a system can then be used to analyze the compliance with the requirements of such standards.

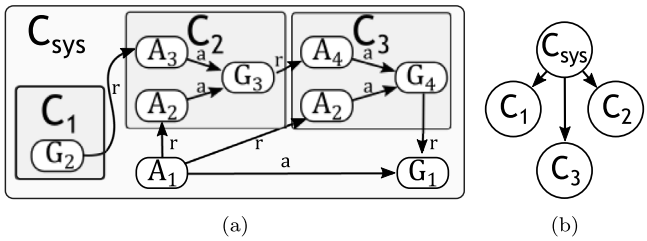


Fig. 3. Graphical representation of a toy specification structure (a), and the corresponding architecture (b).

While Definition 8 defines the possible nodes and edges within a specification structure, it does not impose syntactic constraints that prevent incomplete, or inconsistent specification structures. For example, there could exist assumptions or guarantees that are not intended to be refined by any specification, and consequently not satisfied by any component. For example, in Fig. 3(b), if the intended refinement between $\mathtt{G_2}$ and $\mathtt{A_3}$ would not exist, then such model would represent a system where $(\{A_3, A_2\}, G_3)$ is not satisfied according to Definition 7. Also, the graph of the specification structure, and consequently the specifications within the contracts, might contain directed cycles which can lead to a flawed system implementation. To avoid such cases, additional constraints are introduced.

Definition 9 (*Proper Specification Structure*)**.** A specification structure $\mathfrak{D}$ is *proper* if

(i) for each A_i^j of a contract allocated to a non-root component $C_m \in \mathscr{A}$, there exists an $S \in \mathcal{S}$ and an edge $e^{\mathtt{r}} = (S, A_i^j)$ such that either

(a) $S \equiv A_i^k$ of a contract $K_k \in \mathcal{K}$ and there exists a $C_p \in \mathscr{A}$ such that C_p is the parent of C_m and $\mathtt{allTo}(K_k, C_p)$ or,
(b) $S \equiv G_k$ of a contract $K_k \in \mathcal{K}$ and there exists a $C_p \in \mathscr{A}$ such that C_p and C_m are siblings and $\mathtt{allTo}(K_k, C_p)$.

(ii) for each G_j of a contract allocated to a non-leaf component $C_m \in \mathscr{A}$, there exists an edge $e^{\mathtt{r}} = (G_k, G_j)$ and a component $C_p \in \mathscr{A}$, such that G_k is the guarantee of a contract $K_k \in \mathcal{K}$, C_m is the parent of C_p, and $\mathtt{allTo}(K_k, C_p)$
(iii) $\mathfrak{D}$ does not contain a directed cycle. □

Assuming that the component composition is monotonic (Westman and Nyberg, 2018b), the following theorem is the key idea of CBD and it corresponds to the *dominance relation* from (Graf and Quinton, 2007).

Theorem 1. *Let $\mathscr{A}$ be an architecture with the root C_r, and let $\mathcal{K}$ be a set of contracts such that each $K \in \mathcal{K}$ is allocated to at least one component from $\mathscr{A}$ and contract $K_r \in \mathcal{K}$ is allocated to C_r. Let $\mathcal{K}_{At} \subseteq \mathcal{K}$ be the set of contracts allocated to atomic components $\mathcal{C}_{At} \subseteq \mathscr{A}$. If*

(i) $\forall K \in \mathcal{K}_{At}.\forall C \in \mathcal{C}_{At}.\mathtt{allTo}(K, C) \rightarrow C \blacktriangleright K$,

(ii) $\mathcal{K}$ forms a proper specification structure $\mathfrak{D}$,
(iii) for each edge $e^r = (S_1, S_2)$, it holds that $S_1 \sqsubseteq S_2$,

then it holds that $C_r \blacktriangleright K_r$. □

Proof. The proof can be found in Appendix.

Theorem 1 essentially states that: (i) if the atomic components satisfy the allocated contracts, e.g. software and hardware components, and (ii) if the system design follows the prescribed design principles, here captured as a *proper* specification structure, and (iii) if the *intended* refinement specified by e^r edges actually corresponds to refinement, *then it can be inferred* that the overall system satisfies the allocated contract.

The theory presented so far does not support specifying systems, i.e. composite components, that are configurable. In the next section, the presented CBD framework is extended with product line constructs.

5. Product-line extension of the CBD framework

This section presents the first contribution, which is a product-line extension of the theory presented in Section 3.2 and Section 4. Before the formal definitions, we provide some intuition.

Consider using the presented CBD framework to represent and analyze a configurable system. This implies that the *design* of each system variant would have to be defined as a separate specification structure $\mathfrak{D}$. The upper part of Fig. 4 shows two specification structures of the composite component $\mathtt{C_{sys}}$ for configurations γ_1 and γ_2, which represent two different system variants. For different configurations, $\mathtt{C_{sys}}$ is composed of different components, i.e. $\mathtt{C_{sys}} = \mathtt{C_1} \otimes \mathtt{C_2}$ versus $\mathtt{C_{sys}} = \mathtt{C_1} \otimes \mathtt{C_3}$, and different contracts are allocated to the same components, e.g. $\mathtt{allTo}((\{\mathtt{A_3}\}, \mathtt{G_3}), \mathtt{C_1})$ versus $\mathtt{allTo}((\{\mathtt{A_4}\}, \mathtt{G_3}), \mathtt{C_1})$. Besides the fact that common components and contracts would be duplicated across multiple specification structures, more importantly, the reasoning about whether each variant of C_{sys} satisfies the contract $(\{\mathtt{A_1}\}, \mathtt{G_1})$, would have to be performed per variant. Because the number of configurations, and consequently system variants can be very high, performing per-variant analysis is usually not feasible. The alternative is the approach proposed in the present paper, namely to directly create a *product-line specification structure* and analyze it simultaneously for all configurations, regardless of their number. Such model would correspond to the *integrated platform* from which *variants* of the system are derived. The bottom part of Fig. 4 shows the product-line specification structure that simultaneously expresses the two variants of C_{sys}.

Because a product-line specification structure represents multiple variants, the idea is that if each variant is a *proper specification structure*, then Theorem 1 can be used to infer that the configurable system satisfies the allocated contracts, for all configurations γ. However, to ensure that the design of each variant is a *proper specification structure*, and because a product-line specification structure represents a configurable system, additional constraints must be enforced. In Fig. 4, guarantee $\mathtt{G_3}$ is intended to refine assumption $\mathtt{A_2}$, and component $\mathtt{C_1}$ is intended to satisfy the contract $(\{\mathtt{A_3}\}, \mathtt{G_3})$. Given a configuration γ, to which guarantee $\mathtt{A_2}$ applies, it must be ensured that each of the artifacts $\mathtt{C_1}$, $\mathtt{A_3}$, and $\mathtt{G_3}$ also apply to configuration γ. Also, either assumption $\mathtt{A_3}$ or assumption $\mathtt{A_4}$ should apply to configuration γ, but not both.

To understand the necessity for enforcing additional constraints, consider the first case. If there exists a variant that contains artifacts $\mathtt{C_2}, \mathtt{A_2}, \mathtt{G_2}$, and $\mathtt{G_1}$ but not all of them, that could mean that either: (i) the component $\mathtt{C_2}$ is part of the variant but the contract $(\mathtt{A_2}, \mathtt{G_2})$ does not, or (ii) $\mathtt{A_2}$ and $\mathtt{G_2}$ do not jointly comprise the variant, i.e. the contract $(\mathtt{A_2}, \mathtt{G_2})$ is ill-formed with respect to Definition 8. The described issues are

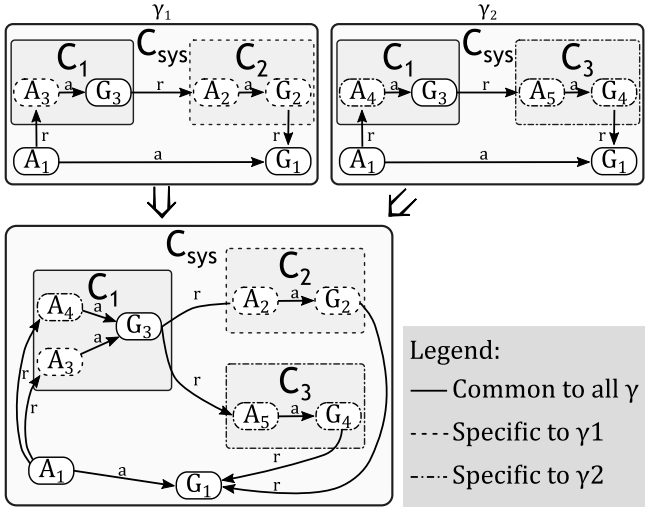


Fig. 4. Per-configuration specification structures (top), and the concept of a product-line specification structure (bottom).

referred to as *configuration mismatches* (El-Sharkawy et al., 2017), and in general they represent a mismatch between the *product line design*, here represented as a CBD model, and the *product line specification*, which is here represented as a variability model.

Besides verifying the absence of configurations mismatches, it is crucial that such verification is performed only once, against the product-line specification structure, and not against parts of the product-line specification structure that represent a particular variant. In this way, the challenge of analyzing billions of variants is avoided. In the following two subsections, the product-line specification structure is formally defined, *presence conditions* are introduced, and the constraints that ensure absence of configuration mismatches are defined.

5.1. CBD model of a configurable system

The definition of a *product-line* specification structure is similar to *specification* structure from Definition 8, with the difference that it simultaneously represents the design of several configurations from Γ. Before formally introducing the product-line specification structure, we define the *product-line architecture* where components can be shared between different system variants, i.e. the architecture is not a *tree* as in Definition 4.

Definition 10 (*PL Architecture*)**.** A *PL architecture* $\widehat{\mathscr{A}}$ is a finite, non-empty set of components organized into a *rooted, directed, acyclic graph* where each *leaf* node is an *atomic* component, and each *non-leaf* node C with children nodes $C_1, \ldots, C_n$ is a *composite component*, such that $C = \bigotimes_{i=1}^{n} C_n$. □

To create a product-line specification structure that represents the design of all configurations from Γ, regardless of their number, it is necessary to define to which configurations from Γ each component and each specification apply to. As discussed above, this is achieved by labeling each specification and each component with a *presence condition*.

Definition 11 (*PL Specification Structure*)**.** Let $\widehat{\mathscr{A}}$ be a PL architecture with the root component C_r. Let $\mathcal{S}$ be a finite, non-empty set of specifications that form the set of contracts $\mathcal{K}$ where each $S \in \mathcal{S}$ is either an assumption or a guarantee of a contract $K \in \mathcal{K}$, and each $K \in \mathcal{K}$ is allocated to at least one component from $\widehat{\mathscr{A}}$. Then, a *PL specification structure* $\widehat{\mathfrak{D}}$ for $\mathcal{S}$ is a tuple $\widehat{\mathfrak{D}} = (\mathcal{N}, \mathcal{E}, \widehat{\mathscr{A}}, p)$, where

(i) conditions (i)–(vi) from Definition 8 hold,
(ii) p is a function that labels each specification $S \in \mathcal{S}$, and each component $C \in \widehat{\mathscr{A}}$, with a presence condition φ according to Definition 3. □

Similarly to the basic CBD framework, we also define a *proper* PL specification structure.

Definition 12 (*Proper PL Specification Structure*)**.** A PL specification structure $\widehat{\mathfrak{D}}$ is *proper* if

(i) condition (i) of Definition 9 holds,
(ii) condition (ii) of Definition 9 holds. □

Unlike Fig. 4 which exemplifies a PL specification structure, Fig. 5 shows a *proper* PL specification structure, the underlying architecture, and examples of presence conditions, for configurable system described by the feature model from Fig. 2(b).

The example represents the design of a toy product-line where $\mathtt{C}_{\mathtt{sys}}$ is composed of $\mathtt{C}_{\mathtt{subSys1}}$ and $\mathtt{C}_{\mathtt{subSys2}}$. The meaning of the visual elements is identical as for Definition 8 with the additional visualization of presence conditions (condition (ii) of Definition 11). The presence conditions are written with respect to the feature model from Fig. 2, where the symbol T denotes that the presence condition is always *true*, i.e. components and specifications labeled with T apply to all configurations from Γ. On the other hand, for example, presence conditions φ_{sys}, φ_1, and φ_2 define that $\mathtt{C}_{\mathtt{sys}}$ is composed of $\mathtt{C}_{\mathtt{subSys1}}$ for all configurations where $f_4 = true$, composed of $\mathtt{C}_{\mathtt{subSys2}}$ for all configurations where $f_5 = true$, and because f_4 and f_5 form an *alternative group* in the feature model, no valid configuration will simultaneously assign the value *true* to both f_4 and f_5. This means that $\mathtt{C}_{\mathtt{sys}}$ exists in different variants with respect to component composition.

According to Definition 10, and in contrast to Definition 4, a single component can be a part of multiple compositions, e.g. $\mathtt{C}_{\mathtt{subSys1}}$ and $\mathtt{C}_{\mathtt{subSys2}}$ execute different software, but they have a common hardware, namely $\mathtt{C}_{\mathtt{HW}}$. Also, because a proper PL specification structure is a superimposition of several configuration-specific specification structures, in the general case, the graph of $\widehat{\mathfrak{D}}$ can contain a directed cycle and therefore the condition (iii) from Definition 9 is not included in Definition 12. For example, in Fig. 5, contract $(\mathtt{A}_5, \mathtt{G}_5)$ is allocated to $\mathtt{C}_{\mathtt{HW}}$ and $\mathtt{C}_{\mathtt{subSys2}}$. Although these are visually represented separately for comprehension, formally the set of edges $\mathcal{E}$ of the proper PL specification structure in Fig. 5 contains the cycle $e^{\mathtt{r}} = (A_2, A_5)$, $e^{\mathtt{r}} = (A_5, A_4)$, $e^{\mathtt{a}} = (A_4, G_4)$, $e^{\mathtt{r}} = (G_4, A_2)$.

As outlined in Section 3.1, artifacts are labeled with presence conditions to define the set of configurations to which the artifacts apply. Then, function *eval* evaluates the presence conditions for a given configuration, and the artifacts whose presence conditions evaluate to *true* are selected, thus obtaining the artifacts that comprise the variant that corresponds to the selected configuration. To be able reason about this process, we formally introduce the concept of a *variant* which is derived from a PL specification structure by removing the artifacts whose presence conditions evaluate to false for a given configuration.

Definition 13 (*Variant*)**.** Let $\widehat{\mathfrak{D}}$ be PL specification structure, and let $\gamma \in \Gamma$ be a configuration. A *variant*, denoted $\mathfrak{D}_\gamma$, is a tuple $\mathfrak{D}_\gamma = (\mathcal{N}_\gamma, \mathcal{E}_\gamma, \mathscr{A}_\gamma)$ where

(i) $\mathcal{N}_\gamma \subseteq \mathcal{N}$ such that $\forall n \in \mathcal{N}_\gamma$ where n represents a specification S, it holds that $eval(p(S), \gamma) = true$,
(ii) $\mathcal{E}_\gamma \subseteq \mathcal{E}$, where for each $e = (S_1, S_2)$ it holds that $eval(p(S_1), \gamma) = eval(p(S_2), \gamma) = true$,
(iii) $\mathscr{A}_\gamma \subseteq \widehat{\mathscr{A}}$ such that $\forall C \in \mathscr{A}_\gamma . eval(p(C), \gamma) = true$,
(iv) components from $\mathscr{A}_\gamma$ form a tree. □

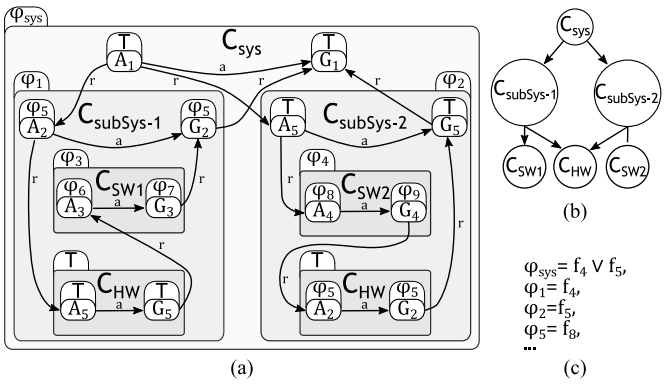


Fig. 5. Example proper PL specification structure (a), the corresponding PL architecture (b), and example presence conditions (c).

A variant $\mathfrak{D}_\gamma$ is not necessarily a specification structure according to Definition 8. For example, in the context of Fig. 5, it might be the case that $eval(\varphi_6, \gamma) = eval(\varphi_7, \gamma) = true$ and $eval(\varphi_3, \gamma) = false$, i.e. for configuration γ contract $(\mathtt{A}_3, \mathtt{G}_3)$ is not allocated to any component. On the other hand, if we could verify that any variant is a specification structure (c.f. Definition 8), and furthermore a *proper* specification structure (c.f. Definition 9), then Theorem 1 could be extended to claim that for any configuration, a configurable system satisfies the allocated contracts.

5.2. Constraints on presence conditions

To ensure that each *variant* is a *proper* specification structure according to Definition 9, we introduce the following conditions, and we refer to them as *invariance with respect to configurations*, hereinafter only *invariant*. In the following definitions, the symbol $\models_\Theta$ represents logical entailment with respect to the set of Boolean formulas Θ of a variability model $\mathfrak{M}$. More formally, $f_i \models_\Theta f_j$ is equivalent to $f_i, \Theta \models f_j$. For example, given Boolean features f_1 and f_2 it holds that $f_1 \not\models f_2$ but in the context of the feature model from Fig. 2(a), where Θ contains formula $f_1 \rightarrow f_2$, then it holds that $f_1 \models_\Theta f_2$.

Definition 14 (*Invariant Contract*)**.** A contract $(\mathcal{A}_j, G_j)$ is *invariant* if

(i) $\forall A_i^j \in \mathcal{A}_j . p(A_i^j) \models_\Theta p(G_j)$,
(ii) $\forall A_i^j \in \mathcal{A}_j . p(G_j) \models_\Theta p(A_i^j)$. □

Less formally, Definition 14 defines the conditions under which each contract is well-formed according to Definition 6 for each configuration.

Definition 15 (*Invariant Allocation*)**.** An *allocation of* a contract $(\mathcal{A}_j, G_j)$ to possibly n different components, is *invariant* if $p(G_j) \models_\Theta \bigvee_{i=1}^{n} p(C_i)$. □

The condition of Definition 15 ensures that if a contract is a part of a variant, then at least one component to which the contract is allocated to, is also a part of the same variant. Note that Definition 15 includes two cases of invariance. The first case is if multiple components are redundant and should satisfy the same contract. In that case, for any γ, presence conditions of multiple components C_i from Definition 15 will evaluate to *true* whenever $p(G_j) = true$. The second case is if different components should satisfy the same contract for different configurations. In that case, for different γ, the presence condition of a single component C_i from Definition 15 will evaluate to *true* whenever $p(G_j) = true$.

Definition 16 (*Invariant Composition*)**.** Composition of n components $C_1, \ldots, C_n$, into $C = \bigotimes_{i=1}^{n} C_i$ is *invariant* if it holds that $\forall C_i \in \{C_1, \ldots, C_n\}.p(C_i) \models_{\Theta} p(C)$. □

Definition 16 ensures that if a component is a part of a variant, then its parent component is a part of the same variant. For example, if the presence condition of $\mathtt{C}_{\mathtt{SW1}}$ evaluates to *true*, then the presence condition of $\mathtt{C}_{\mathtt{subSys1}}$ should evaluate to *true*.

Given a PL specification structure $\widehat{\mathfrak{D}}$ with invariant contracts, allocation, and composition, instantiating $\widehat{\mathfrak{D}}$ for a configuration γ, results in a specification structure according to Definition 8.

Theorem 2. *Let Γ be a set of valid configurations, and let $\widehat{\mathfrak{D}}$ be a proper PL specification structure. If*

(i) each contract is invariant,
(ii) allocation of contracts to components from $\widehat{\mathscr{A}}$ is invariant,
(iii) composition of child into parent components in $\widehat{\mathscr{A}}$ is invariant,

then each variant $\mathfrak{D}_\gamma$, is a *specification structure.* □

Proof. The proof can be found in Appendix.

Theorem 2 establishes the conditions under which a variant is a specification structure. However, using Theorem 1 requires that a variant is a *proper* specification structure. The following theorem defines additional conditions under which each variant is a *proper* specification structure according to Definition 9. Note that one of the conditions is that the graph of $\widehat{\mathfrak{D}}$ does not contain a directed cycle, and consequently, no variant $\mathfrak{D}_\gamma$ contains a directed cycle as required by Definition 9. Although the cycles such as the one described in Fig. 5 can be avoided, e.g. by allowing that a particular contract can be allocated only to components at the same architectural level, in the general case, detecting cycles may require the exploration of the complete graph of $\widehat{\mathfrak{D}}$, which can be done efficiently by using known algorithms, e.g. *depth-first search* (Rosen and Krithivasan, 2012, p. 787).

Theorem 3. *Let Γ be a set of valid configurations, and let $\widehat{\mathfrak{D}}$ be a proper PL specification structure such that each variant $\mathfrak{D}_\gamma$ is a specification structure. If*

(i) for each $A_i^j \in \mathcal{S}$, and edges $e^r = (S_1, A_i^j), \ldots, (S_n, A_i^j)$ it holds that $p(A_i^j) \models_{\Theta} \bigvee_{k=1}^{n} p(S_k)$,
(ii) for each $G_j \in \mathcal{S}$, and edges $e^r = (G_1, G_j), \ldots, (G_n, G_j)$ it holds that $p(G_j) \models_{\Theta} \bigvee_{k=1}^{n} p(G_k)$
(iii) $\widehat{\mathfrak{D}}$ does not contain a directed cycle,

then each $\mathfrak{D}_\gamma$ is proper *specification structure.* □

Proof. The proof can be found in Appendix.

A direct consequence of Theorems 2 and 3 in conjunction with Theorem 1 is the following corollary.

Corollary 1. *Let $\widehat{\mathfrak{D}}$ be a proper PL specification structure for a set of specification $\mathcal{S}$ that form a set contracts $\mathcal{K}$. Let $K_r \in \mathcal{K}$ be the contract allocated to the root component $C_r \in \widehat{\mathscr{A}}$, and let $\mathcal{K}_{At} \subseteq \mathcal{K}$ denote the set of contracts allocated to the set of atomic components $\mathcal{C}_{At} \subseteq \widehat{\mathscr{A}}$. Let Γ be a set of valid configurations, defined by a variability model $\mathfrak{M}$. If*

(i) premises of Theorem 2 hold,
(ii) premises of Theorem 3 hold,
(iii) for each edge $e^r = (S_1, S_2)$, it holds that $S_1 \sqsubseteq S_2$,
(iv) $\forall K \in \mathcal{K}_{At}.\forall C \in \mathcal{C}_{At}.\mathtt{allTo}(K, C) \rightarrow C \blacktriangleright K$,

then it holds that $C_r \blacktriangleright K_r$ for each $\gamma \in \Gamma$. □

Corollary 1 summarizes the main results so far. If a proper PL specification structure is considered to be a representation of a product line, i.e. the *integrated platform* from which *variants* are derived, then conditions (i)-(iii) of Corollary 1 must be satisfied by the design, and condition (iv) must be satisfied by the implementation, in order to be able to claim that for configuration the contract K_r is satisfied. As will be shown in Section 7, providing evidence that the premises of Corollary 1 hold is the basis for creating assurance-case arguments.

However, before proceeding to the creation of assurance-case arguments, the applicability of the CBD framework to a realistic setting is evaluated.

6. Applying the CBD framework to a real product line

This section presents an application of the extended CBD framework with the objective to understand if it can be used to represent and analyze a real, safety-critical product-line. The *hypothesis* is that applying the CBD framework to a real product-line will either lead to the creation of *proper* PL specification structure, thus enabling the application of Corollary 1, or one or more conditions of the CBD framework will not be satisfied, and this might lead to recommendations to improve the product-line design or implementation. Recall that the purpose of the CBD framework *is not* to be used as a concrete modeling-framework for different artifacts within the product line. Consequently, we do not attempt to evaluate if it is possible, and what is the required effort, to adopt the CBD framework in everyday engineering. Rather, a CBD model is a *formal representation* that is constructed *on-demand* for analysis purposes, i.e. the CBD framework is intended to *enable preferably automated analysis* of arbitrary product lines, which are developed using tools and notations that best suite everyday activities within a particular company.

The questions that this section seeks to answer are:

Q1 Given a real product line with a variety of concrete artifacts in different notations, is it possible to represent such product line as a CBD model?
Q2 If the CBD representation of a product line violates some conditions of the CBD framework, can these violations be translated into actionable engineering insights, or are these a consequence of formalization of the CBD-framework?

6.1. Methodology

Given the open-ended nature of the above questions, we adopt the structure of an *exploratory case study* (Runeson et al., 2012) to reach the answers. In accordance with (Runeson et al., 2012), the phases of the performed case study are shown Fig. 6. The product line used as the case is a *real, safety-critical* product line from the heavy vehicle manufacturer Scania.

Selecting the case. The selected case is the *Fuel-Level Display* (FLD) system, whose functionality is available in each Scania vehicle. This system was selected for several reasons. Firstly, it is a typical system from the automotive domain. It is implemented across several *Electronic Control Units* (ECUs) within the vehicle, its behavior directly depends on the configuration selected by the customer, and the artifacts within the product-line are created in a V-like development process. Also, the system is *safety-critical* (internal Scania rating corresponds to ASIL B), which means that the artifacts should be of sufficient quality for a rigorous analysis. Last, but not the least, a practical consideration is that parts of the FLD system have been published earlier, thus Scania was willing to allow a publication based on the FLD system. It should be noted that the analyzed version of the FLD system, with the corresponding artifacts, was developed around 2015.

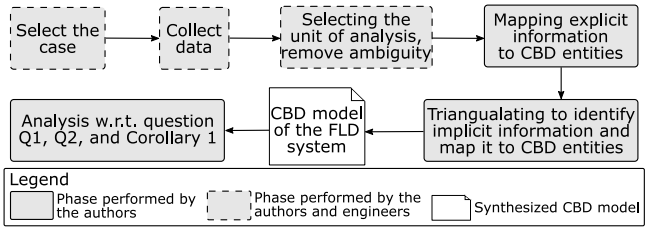


Fig. 6. Phases of the exploratory case study.

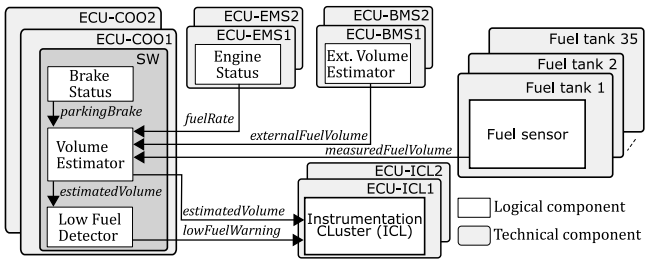


Fig. 7. Illustration of the FLD system architecture.

Selecting the unit of analysis. After selecting the case, the authors collected the artifacts of the FLD system with the help of two Scania engineers who pointed out the location and the correct versions of the artifacts. Given the overall system, the authors selected a part of the system to be the *unit of analysis* (Runeson et al., 2012) such that the unit is comprehensive enough to apply the CBD framework, but small enough for a manual analysis and subsequent presentation of the results. This activity has revealed ambiguities within the analyzed artifacts, primarily among the natural-language requirements, and another consultation was made with the same Scania engineers in order to disambiguate the artifacts and also to confirm that the unit of analysis actually represents a part of the *end-to-end* functionality of the FLD system.

Executing the study. The actual synthesis of the CBD model was performed without the help of the engineers because our long term vision is that a CBD model of a system can be created automatically. First, explicit entities within the collected artifacts were mapped to the concepts of the CBD framework, i.e. components, specifications, presence conditions. Then, the explicit traceability links within the artifacts were mapped to the relations of the CBD framework, i.e. allocated to, refines, composition, assumption of. At this point, the CBD model *was not* even a PL specification structure because the development practice in Scania was to allocate requirements *only* to *logical components*, which represent the overall system, and to the main software components. Therefore, the requirements and the traceability links for the implementation components at various abstraction levels were missing. To identify the implicit information, we relied on *triangulation* (Runeson et al., 2012) between the available artifacts, with respect to the conditions of the CBD framework. Triangulation is a frequently used method in qualitative research where the same problem is analyzed by using different data-sources, by using different methods etc. Triangulation is typically performed to increase the *validity* of the results, and to minimize the risk of introducing bias. In the context of the present paper, triangulation was performed over several data sources to ensure the validity of inferred information. For example, descriptive parts of the available *requirements documents* contained informal text with implicit references to the ECUs that should implement the requirements within the documents. Therefore, this text could have been used as the source to infer the missing *allocation relations*. Instead, triangulation was performed to ensure the validity of the inferred information. Firstly the requirements within the documents were analyzed to identify the variables that the requirements constrain. Then, these variables would be identified in the source code of an ECU, which would lead to the software function that is producing this value. Only if the value type and range from a requirement and from the software function would match, we would infer that this particular specification is allocated to the particular software function.

After this step, the final CBD model was obtained, and it was analyzed with respect to the conditions of Corollary 1 to answer the formulated questions. The following sections present the FLD system in more detail, then we present the obtained CBD model, and then discuss the answers to the formulated questions in the context of lessons learned during the creation of the CBD model.

6.2. The FLD system and the analyzed artifacts

The functionality of the FLD system is twofold. Firstly, the FLD should ensure that the fuel volume displayed to the driver, corresponds to the actual fuel volume in the fuel tank. Secondly, if the fuel volume falls below a certain threshold, the FLD should display a warning to the driver. The FLD is safety-critical because if the displayed fuel volume is higher than the actual one, the driver might continue driving until unexpectedly running out of fuel. Consequently, the engine will shut-down, leading to loss of power-steering which is essential for heavy-vehicle steering.

Table 1 shows the artifacts that were analyzed in order to extract the information about the FLD system while Fig. 7 shows the main logical and technical components of the FLD system, together with information flows between the components. The model in Fig. 7 is created primarily from the data from the *logical architecture* and the *product structure* which contains all technical components of the FLD system. The specified behavior of the different components, as defined in analyzed requirements documents, is as follows. The component `Volume Estimator` estimates the current fuel-volume, given the fuel-level sensor readings, and the current rate of fuel consumption. The estimated fuel volume is an input to the *Instrumentation CLuster* (`ICL`), which indicates the estimated volume to the driver. Note that because `ICL` is bought from a supplier, from the Scania perspective this is a logical component, implemented by a whole ECU also called `ICL`. The estimated fuel volume is also an input to the `Low Fuel Detector`, which calculates if the fuel volume is below a threshold. If so, a low fuel warning is sent to the `ICL`. To reliably detect re-fueling, the `Volume Estimator` needs the information about the parking-brake status.

Although the FLD is installed in each Scania vehicle, the system itself is highly-configurable and this is witnessed by the fact that almost all artifacts in Table 1 contain presence conditions. For example, the presence condition of the overall system defines that the FLD system can build into a truck or a bus, and for buses with more than one fuel tank, the fuel volume in additional tanks is estimated by the component `External Volume Estimator`. In such cases, the component `Volume Estimator` merges its estimate with the external one, and calculates the total estimate. Moreover, the logical components from Fig. 7 are allocated differently for different variants. The allocation of the logical components from Fig. 7 applies only to variants that use liquid fuel, while for gas-powered vehicles, `Volume Estimator` and `Low Fuel Detector` are allocated to the *Engine Management System* (`EMS`).

Regarding technical components, the ECUs also differ depending on the configuration of the vehicle. For example, `COO` ECU

Table 1
Analyzed artifacts of the FLD.

Artifact	Format	Content
Logical architecture	Custom graphical format	Logical components with presence conditions, allocated to ECUs, CAN signals between components
Requirements document	Controlled natural language	Functional, functional-safety requirements with presence conditions of the FLD system
Requirements document	Controlled natural language	Per logical component, functional, functional-safety requirements with presence conditions.
Source code	C lang.	Implementation of logical components with configuration variables
Configuration database	Custom textual format	Presence conditions for different values of configuration variables
Product structure	Custom textual format	Hardware, electrical, electromechanical components of the FLD system with presence conditions
Verification documents	Custom textual template	Verification results of SW and HW components, ECUs, and the FLD system
Variability model	Custom textual format	Around 50.000 features with corresponding Boolean constraints

can be built with a *basic* or an *extended* hardware, both bought from suppliers, thus giving rise to COO1 and COO2. Furthermore, there are 35 different fuel tanks, with different shapes and volume. Also, there are 16 different types of fuel-level sensors, which implement the logical component Fuel Sensor, and that can be combined with the different fuel tanks. The variability of the FLD system is also heavily reflected in the source code through the so-called *configuration variables* whose values are set at *load-time* (Rosenmüller et al., 2011) from the *configuration database* depending on the vehicle configuration. For example, the C-language implementation of the Volume Estimator is essentially a large switch statement where the different cases are executed depending on the values of configuration variables. Also, the invocation of whole software components, e.g. Low Fuel Detector and External Volume Estimation, is controlled by the values of configuration variables. In summary, the estimated number of unique FLD variants is around 24,000.

In relation to Fig. 7, the unit of analysis of the FLD system includes the COO ECU, with two different hardware and a single software, two variants of the ICL ECU, namely ICL1 and ICL2, and two fuel-tank assemblies that can be installed individually or jointly, and which include four different fuel tanks and two different sensors. In total, the unit of analysis include 24 variants of the FLD system, and it includes various types of components, at different abstraction levels, that implement the basic functionality of reading the sensor value, estimating the fuel volume, and displaying it to the driver.

6.3. Synthesis of the CBD model

Given the unit of analysis, the obtained CBD model of the FLD system is shown in Fig. 8, while the corresponding specifications are shown in Table 2. The presence conditions of the obtained CBD model are discussed later in Section 6.3.1.

All components in Fig. 8 were directly identified from the *logical architecture*, *product structure* and *source code* since these were organized in terms of SCANIA equivalents of CBD components. Except in the source code, component C_{FLD} exists in each of these artifacts because it represents the FLD system. Components C_{ICL1}, C_{ICL2}, C_{TankL}, C_{TankR} and all of their subcomponents were identified from the *product structure*. Also, C_{COO}, $C_{BasicHW}$ and $C_{ExtendHW}$ were also identified from the product structure. Component C_{COOSW} represents the complete source code of COO, and components $C_{Fuel20ms}$, C_{R2Perc}, and C_{Est} were identified from the source code (a software-architecture artifact did not exist). Note that although COO software contains many application components which run on a common infrastructure software, here we represent only the application component of the FLD system, namely the $C_{Fuel20ms}$, where the name indicates how often the application is executed. Also note that $C_{Fuel20ms}$ implements the logical component Volume Estimator from Fig. 7. Furthermore, although C_{ICL1}, and C_{ICL2} are complete ECUs, because they are bought from suppliers, from the SCANIA perspective they are only atomic components.

When it comes to assumptions $A_1 - A_{11}$ and guarantees $G_1 - G_{14}$ in Table 2, these are the requirements from requirements documents, obtained after the disambiguation with SCANIA engineers. However, according to the SCANIA practices, the analyzed requirements were defined only for the FLD system, represented by C_{FLD} component, and for the logical components. Namely, these are Volume Estimator, implemented by $C_{Fuel20ms}$, Fuel sensor implemented by C_{Sens1} or C_{Sens2}, and ICL implemented by C_{ICL1} or C_{ICL2}. Consequently, the specifications of these components were to a large extent explicitly allocated within the requirements documents.

Individual requirements (c.f. Table 2) primarily defined how the different components should behave, i.e. which properties should be *guaranteed*, thus the majority of the requirements were interpreted as guarantees, namely G_1, G_4–G_9, G_{11}, and G_{14}. For G_1, G_4, G_9, G_{11}, and G_{14}, the requirements documents contained *explicit* allocation information. For guarantees G_5–G_8, the allocation was inferred by identifying the values that a guarantee constrains, e.g. *percent fuel level* in G_7, and then finding the component which produces this value, e.g. the C-function represented by C_{R2Perc}. Because a contract contains exactly one guarantee (c.f. Definition 6), identifying the guarantees meant that the corresponding *contracts* are declared.

When it comes to the assumptions of contracts, few requirements from the analyzed documents were clearly marked as necessary precursors for other requirements, i.e. *assumptions*, and they were explicitly allocated. These assumptions are A_2, A_3 A_9–A_{11}. Assumptions A_2 and A_3 were a part of the requirements document for the overall system, and they were allocated to the ECU that implements the logical component Volume Estimator, which is C_{COO}. Also, the requirements documents contained references to the data-sheets of sensors, which in turn declared the expected environment in which they operate, i.e. A_{10} and A_{11}. Note that A_{10} and A_{11} abstract away the details of the assumptions to reduce clutter. Finally, because ICL is bought from a supplier, C_{ICL1} and C_{ICL2} were accompanied by a data-sheet that explicitly declared assumptions.

For all the considered components, and all requirements within the documents, except for the requirements within the data-sheets of components bought from suppliers, the presence conditions were explicitly defined. In the analyzed artifacts they were called *configuration parameters*, and they were actually written according to the grammar from Definition 3, but with different symbols, e.g. instead of symbol $\wedge$, symbol && was used.

As can be seen from Fig. 8, the explicit requirements within the requirements documents were insufficient to identify the contracts for all considered components. Moreover, there were no traceability links between the requirements of different documents, which would correspond to the intended refinement captured by e^r edges in a PL specification structure. However, this was expected because the FLD system was not developed according to CBD principles. Fortunately, this information was possible to infer by triangulating between the available artifacts, as described by the following subsection.

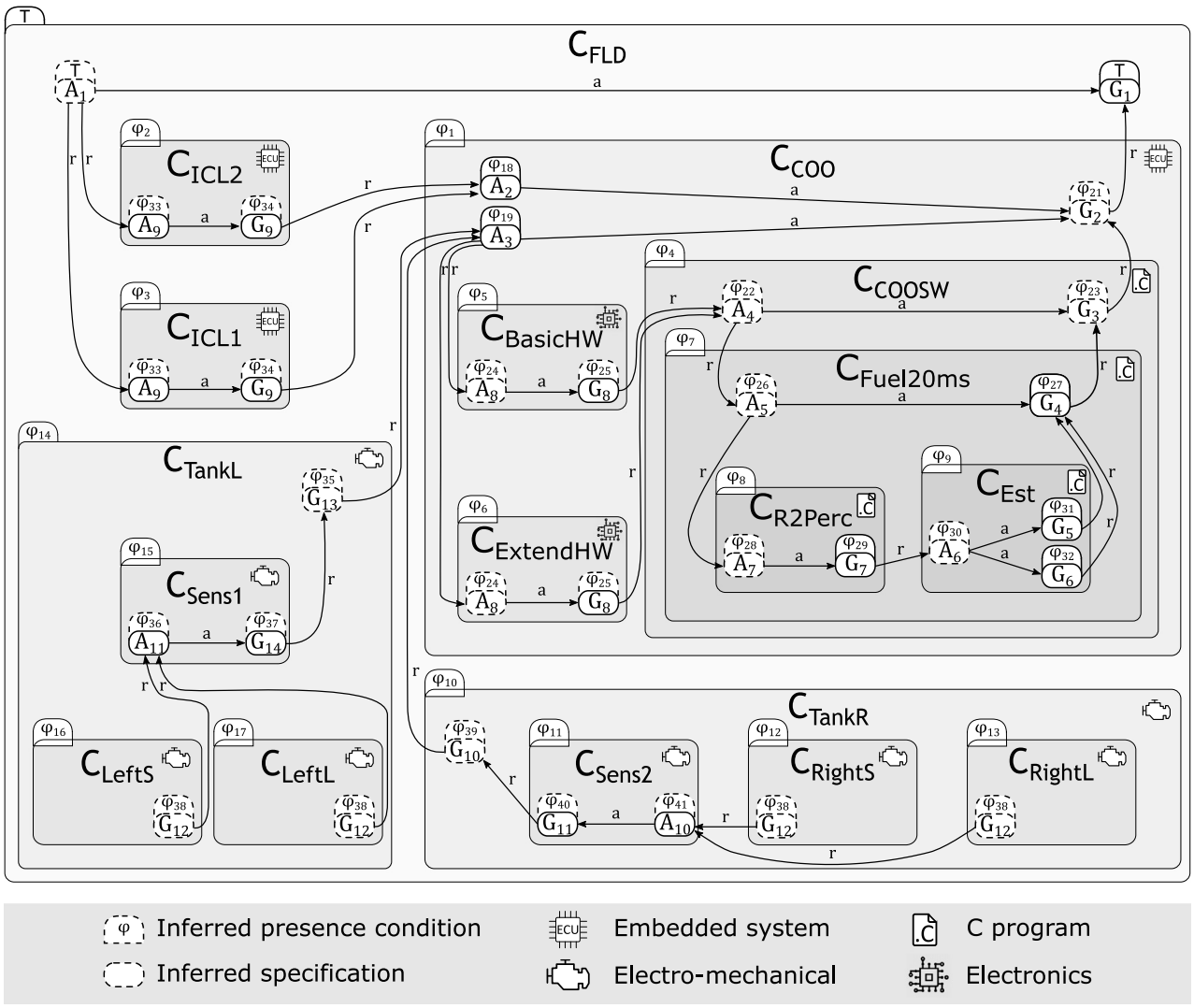


Fig. 8. CBD model of the FLD system for a considered unit of analysis.

Table 2
Assumptions and guarantees from contracts in Fig. 8. Underlined A's and G's were declared in the analyzed documents.

$\underline{G_1} \equiv G_2 \equiv G_3$	The *indicated fuel level* shall not deviate more than ±5% from the actual fuel volume AND The *indicated fuel level* is indicated immediately when the vehicle is switched on AND In the case of a fault mode, the *indicated fuel level* shall be equal to *zero*.
$\underline{A_9} \equiv A_1$	The ignition key is in the *ON position*.
$\underline{A_3} \equiv A_8$	The *sensor resistance value* correspond to datasheet specification.
$\underline{G_9} \equiv \underline{A_2}$	The *indicated fuel level* shall be displayed in the ICL as a bar-graph, and controlled by *estimated fuel volume* [0%–100%].
$\underline{G_4}$	The *estimated fuel volume* corresponds to the *measured fuel volume* AND The *estimated fuel level* shall be transmitted over CAN every 2s from when the C_{COO} starts AND When the fuel level sensor has status *error* or *not available*, then the *estimated fuel level* shall be *zero*.
$\underline{G_5}$	The *estimated fuel level* corresponds to the *measured fuel volume* according to Table X, or *estimated fuel level* value equals *zero*.
$\underline{G_6}$	The *estimated fuel level* corresponds to the *measured fuel volume* according to Table Y, or *estimated fuel level* value equals *zero*.
$\underline{G_7} \equiv A_6$	The *percent fuel level* shall range from 0% to 100%.
$\underline{G_8} \equiv A_4 \equiv A_5 \equiv A_7$	The *measured fuel level* corresponds to the *current sensor resistance* according to the formula $V = (R_{sens} \div (R_{sens} + 10\ \text{K}\Omega)) * 5\ \text{V}$.
$\underline{G_{14}} \equiv G_{13}$	Sensor resistance ranges linearly from 10 Ω to 180 Ω.
$\underline{G_{11}} \equiv G_{10}$	Sensor resistance ranges from 10 Ω to 182 Ω in 26 equidistant, discrete steps.
$\underline{A_{10}} \equiv \underline{A_{11}} \equiv G_{12}$	The electro-mechanical properties of the fuel tank are according to the fuel-sensor datasheet.

6.3.1. Inferring missing information

Because the ideal outcome of the exploratory case study is to create a proper PL specification structure that can be used for the creation of an assurance case, and because our long-term vision is that this process can be automated, the missing information was inferred by triangulating between the available artifacts. Therefore, it was important to *avoid* introducing any new information. This means that the inference steps are *conservative*, and that they build a *best guess* CBD model, which can then be refined if needed.

Missing Guarantees. Regardless of the actual content of a specification that is declared to be a guarantee, there are two conditions of the CBD framework that guarantees must satisfy. Firstly, if the guarantee G belongs to a contract that is allocated to a composite component C, then a CBD model must contain an edge $e^{\mathrm{r}} = (G', G)$ where G' is a guarantee of a contract allocated to a subcomponent of C. (c.f. Definition 9(ii)). Secondly, such e^{r} edge must correspond to specification refinement (c.f. Corollary 1(iii)), e.g. to logical entailment. These two conditions jointly ensure that the specified behavior of a composite component is *a consequence* of the specified behavior of its subcomponents.

Guided by these two conditions, first, for each component that did not have a contract with a guarantee allocated, a guarantee was declared, i.e. guarantees $\mathtt{G}_2$, $\mathtt{G}_3$, $\mathtt{G}_{10}$, $\mathtt{G}_{12}$, $\mathtt{G}_{13}$.

Secondly, to satisfy condition Definition 9(ii), e^{r} edges were declared between guarantees $\mathtt{G}_1$ through $\mathtt{G}_6$, between $\mathtt{G}_{14}$ and $\mathtt{G}_{13}$, and between $\mathtt{G}_{11}$ and $\mathtt{G}_{10}$. For the guarantees $\mathtt{G}_1$ through $\mathtt{G}_6$, regardless of their content, there was no alternative way to declare the e^{r} edges. For the edges $e^{\mathrm{r}} = (\mathtt{G}_{14}, \mathtt{G}_{13})$ and $e^{\mathrm{r}} = (\mathtt{G}_{11}, \mathtt{G}_{10})$, an alternative was to state that guarantee $\mathtt{G}_{12}$ should refine $\mathtt{G}_{13}$ and $\mathtt{G}_{10}$. However, this would mean that $\mathtt{C}_{\mathtt{Sens1}}$ and $\mathtt{C}_{\mathtt{Sens2}}$ do not contribute to the behavior of $\mathtt{C}_{\mathtt{TankL}}$ and $\mathtt{C}_{\mathtt{TankR}}$, and consequently the explicit assumption $\mathtt{A}_3$ would not be refined by any specification.

Thirdly, the actual content of the guarantees was inferred, such that condition Corollary 1(iii)) was satisfied. The special case in which refinement *trivially holds*, regardless of how refinement is defined, is when two specifications are *equivalent*. This principle was adopted to avoid adding any new information, i.e. the content of the inferred guarantees was declared equivalent to the existing requirements from the analyzed requirements documents. For example, $\mathtt{G}_2 \equiv \mathtt{G}_1$ practically means that the functionality of the FLD system, i.e. the fact that the estimated fuel volume corresponds to the actual fuel volume, is a consequence of the functionality of the COO ECU, whose software computes the estimated fuel volume.

Missing assumptions. Any assumption within a CBD model must satisfy two conditions. Firstly, except the overall system assumption, for each assumption A_i of a contract allocated to component C, the CBD model must contain an edge $e^{\mathrm{r}} = (S, A_i)$ where S is either an assumption of a parent of C, or a guarantee of a sibling of C (c.f. Definition 9(i)). Secondly, this edge must correspond to specification refinement (c.f. Corollary 1(iii)).

Guided by these conditions, firstly an assumption was declared for each contract without an assumption. Exceptions are electro-mechanical components $\mathtt{C}_{\mathtt{TankL}}$, $\mathtt{C}_{\mathtt{TankR}}$, $\mathtt{C}_{\mathtt{LeftS}}$, $\mathtt{C}_{\mathtt{LeftL}}$, $\mathtt{C}_{\mathtt{RightS}}$, $\mathtt{C}_{\mathtt{RightL}}$ because their assumptions would correspond to acceptable types of fuels, or allowed fuel pressure, and these were deemed out of scope. Unlike for missing guarantees, the e^{r} edges and the content of newly declared assumptions were inferred simultaneously.

Assumption $\mathtt{A}_2$, $\mathtt{A}_3$, $\mathtt{A}_6$, $\mathtt{A}_{10}$, and $\mathtt{A}_{11}$ are the assumption intended to be refined by guarantees. Regarding $\mathtt{A}_2$, it was explicitly equivalent to $\mathtt{G}_9$ and an e^{r} edge was asserted. This is a great example of *contract thinking* in industry where one company assumes a component that satisfies $\mathtt{A}_2$, and then outsources the development of such component but from the point of view of the supplier this is a guarantee, namely $\mathtt{G}_9$. Regarding $\mathtt{A}_3$, the content of $\mathtt{A}_3$ clearly could only be refined by a guarantee of a sensor, and because the guarantees of sensors were equivalent to guarantees of $\mathtt{C}_{\mathtt{TankL}}$ and $\mathtt{C}_{\mathtt{TankR}}$, the corresponding e^{r} edges were asserted. Regarding $\mathtt{A}_6$, the inspection of the source code revealed that the input of C-function $\mathtt{C}_{\mathtt{Est}}$ was the output of function $\mathtt{C}_{\mathtt{R2Perc}}$. Therefore, the assumption $\mathtt{A}_6$ was concluded to be equivalent to guarantee $\mathtt{G}_7$ and the edge $e^{\mathrm{r}} = (\mathtt{G}_7, \mathtt{A}_6)$ was added, in accordance with Definition 9–(i). Regarding $\mathtt{A}_{10}$, $\mathtt{A}_{11}$, and corresponding e^{r} edges, the data-sheet of the two sensors explicitly stated that the corresponding fuel tanks must refine their assumptions.

Table 3
A subset of the presence conditions from Fig. 8. Underlined φ's were declared in the analyzed artifacts.

$\underline{\varphi_1}$	$(f_{2482-1} \vee (f_{2482-2} \wedge f_{1940-2}) \vee (f_{2482-2} \wedge f_{1940-1})) \wedge (f_{1837-1} \vee (f_{1837-19} \wedge f_{579-26}))$
$\underline{\varphi_4}$	$\varphi_1 \wedge f_{1323-5}$
$\varphi_7 = \varphi_9$	$(f_{1-1} \vee f_{1-2}) \wedge \neg f_{520-2}$
$\underline{\varphi_5}$	$(f_{2482-1} \vee (f_{2482-2} \wedge f_{1940-1}) \vee (f_{2482-2} \wedge f_{1940-1})) \wedge (f_{1837-1} \vee (f_{1837-19} \wedge f_{579-26})) \wedge ((f_{37-1} \wedge f_{2519-1}) \vee (f_{37-3} \wedge f_{2519-1}) \vee f_{37-5})$
$\underline{\varphi_6}$	$(f_{2482-1} \vee (f_{2482-2} \wedge f_{1940-1}) \vee (f_{2482-2} \wedge f_{1940-1})) \wedge (f_{1837-1} \vee (f_{1837-19} \wedge f_{579-26})) \wedge ((f_{37-2} \wedge f_{2519-1}) \vee (f_{37-4} \wedge f_{2519-1}) \vee f_{37-5})$
$\varphi_{24} = \varphi_{25}$	$\varphi_5 \vee \varphi_6$
$\varphi_{23} = \underline{\varphi_{27}}$	$(f_{1-1} \vee f_{1-2}) \wedge f_{520-1}$
$\underline{\varphi_{31}}$	$f_{3431-26} \wedge f_{3706-29} \wedge f_{4323-26}$
$\underline{\varphi_{32}}$	$f_{3431-26} \wedge f_{3706-28} \wedge f_{4323-26}$

Regarding the remaining assumptions, they have an incoming e^{r} edge from another assumption, according to Definition 9–(i). Regarding $\mathtt{A}_7$, the corresponding guarantee $\mathtt{G}_7$ guarantees the value *percent fuel level*. By inspecting the source code, component $\mathtt{C}_{\mathtt{R2Perc}}$ takes as input the *measured fuel value* from the analogue-to-digital converter which is a part of components $\mathtt{C}_{\mathtt{BasicHW}}$ and $\mathtt{C}_{\mathtt{ExtendHW}}$. Consequently, there should exist an e^{r} edge from $\mathtt{G}_8$ to $\mathtt{A}_7$, but such edge is not allowed according to Definition 9–(i). Instead, the allowed edges are from $\mathtt{A}_5$ to $\mathtt{A}_7$ and from $\mathtt{A}_4$ to $\mathtt{A}_5$. Then, because $\mathtt{A}_4$ belongs to a contract that is allocated to a sibling component of $\mathtt{C}_{\mathtt{BasicHW}}$ and $\mathtt{C}_{\mathtt{ExtendHW}}$, the edge $e^{\mathrm{r}} = (G_8, A_4)$ can be asserted and also the content of these guarantees is defined as $\mathtt{G}_8 \equiv \mathtt{A}_4 \equiv \mathtt{A}_5 \equiv \mathtt{A}_7$. Similar reasoning was applied to infer the content and the edges involving $\mathtt{A}_8$. Finally, assumption $\mathtt{A}_1$ was inferred because the explicit assumption $\mathtt{A}_9$ refers to the ignition key of the vehicle, and because the ignition key is not a component of the FLD system, then this assumption can only be refined by a specification allocated to a component outside the FLD system. In other words, assumption $\mathtt{A}_9$ must be an assumption of the overall FLD system, i.e. $\mathtt{A}_1$.

Missing presence conditions. Besides the missing presence conditions of specifications of supplier-bought components, the presence conditions of inferred assumptions and guarantees had to be inferred. Table 3 shows examples of *identified and inferred* presence conditions. The full list is omitted because some expressions are very large, e.g. φ_{15} contains over 45 terms.

Missing presence conditions of specifications that are allocated to electronic and electro-mechanical components were inferred to be equivalent to the presence conditions of these components because the behavior of such components is unchangeable. When the same specification was allocated to several electronic or electro-mechanical components, the presence conditions was declared to be a disjunction of the presence conditions of different components. In this way, presence conditions φ_5, φ_{24}, φ_{25}, and $\varphi_{33} - \varphi_{41}$ were inferred. For example, as Table 3 shows, presence conditions φ_{24} and φ_{25} were set to $\varphi_5 \vee \varphi_6$. The remaining presence conditions, namely φ_{22}, φ_{23}, and $\varphi_{26} - \varphi_{30}$ were inferred to satisfy conditions of Theorem 3, i.e. to ensure that if a specification S_1 is intended to be refined by another specification S_2, then the presence condition of S_2 should entail the presence condition of S_1. To avoid adding new information, in the same way as for assumptions and guarantees, this was done by asserting that $p(S_1) = p(S_2)$. For example, as Table 2 shows, φ_{23} was inferred to be equivalent to φ_{27}.

6.4. Results

In this section we answer the formulated questions. The answer to each question is contextualized with the lessons learned during the synthesis of the CBD model.

6.4.1. Answering Q1

When it comes to question Q1, the answer is that in general it was possible to represent a real, industrial product-line as a CBD model, but several inference steps had to be performed. On the one hand, this was expected, because the presented CBD framework was not used when the FLD system was developed. The inference steps were also expected because the CBD model represents the complete FLD system, from the most abstract components to the detailed software components, and such representation of the FLD system did not exist. To a good extent, the inferences were needed because of the practices in Scania, namely because requirements were written for logical components, which are then interpreted by developers to create the technical components which implement the logical components. Using the terminology of the domain safety-standard ISO 26262, the practice was to explicitly declare *functional safety requirements*, but not the *technical* and *software* safety requirements. Given that since 2018, when a version of ISO 26262 applicable to trucks was released, the FLD system was brought closer to the guidelines of ISO 26262. Therefore, applying the CBD framework to the newest version would require less inference effort.

On a positive note, guided by the conditions of the CBD framework, the necessary inferences were possible to perform only by triangulating between the available artifacts. Out of tens of different conditions, it was sufficient to consider only five of them. These are the two conditions that define the allowed relations between assumptions and guarantees (c.f. Definition 9(i) and Definition 9(ii)), the two conditions that define the required entailment between presence conditions (c.f. Theorem 3), and condition (iii) of Corollary 1. Also note that given our long term vision to create CBD model in an automated fashion, the CBD model was synthesized without any manual effort by domain experts. Although this means that the synthesized CBD model is a conservative, best-guess model, it is important to note that even if a system, such as the FLD system, is deployed and currently in the maintenance phase, it is possible to create an initial CBD model by using the presented inference steps. Then, the obtained CBD model can be used a starting point for further refinement by domain experts.

6.4.2. Answering Q2

Regarding question Q2, first it must be considered if the model in Fig. 8 satisfies all conditions of the CBD framework. The synthesized CBD model is a proper PL specification structure, according to Definition 12, as required by Corollary 1. The model also satisfies condition (iii) of Theorem 3, i.e. the model does not contain directed cycles. Furthermore, the FLD system satisfies condition (iv) of Corollary 1 because for each atomic component, there was a positive verification result in the verification documents. Note that because the FLD system was developed in a non-compositional way, additional verification was performed for integration of SW and HW component into ECUs, and for the integrated FLD system.

However, when it comes to the remaining conditions of Corollary 1, the situation is less favorable. The model in Fig. 8 does not satisfy condition (iii) of Corollary 1, i.e. it is not the case that each $e^{\mathtt{r}}$ edge corresponds to specification refinement. For example, even without a formal analysis it is clear that $\mathtt{G}_4$ does not refine $\mathtt{G}_3$ because $\mathtt{G}_3$ guarantees that a value of *indicated fuel level* is available *immediately* when the vehicle is switched on, but $\mathtt{G}_4$ produces a value of *estimated fuel level* that can be outputted to $\mathtt{C}_{\mathtt{ICL1}}$ or $\mathtt{C}_{\mathtt{ICL2}}$ every 2s. Similar problems exist between $\mathtt{G}_{13}$ or $\mathtt{G}_{10}$ and $\mathtt{A}_3$. Given the fact that the verification documents showed positive testing results, and the fact the FLD system has been deployed for years, the most probable explanation is that during the implementation these inconsistencies among the specifications were identified and corrected, but the modifications were not propagated back to the requirements. If the CBD framework had been used at the time of development, these inconsistencies would have been identified before the implementation phase. Detecting inconsistent and incomplete system design early is particularly important for highly critical systems whose development is expensive.

Regarding the remaining conditions of Theorems 2, and 3, it is unclear if they are satisfied. Because these conditions include the corresponding variability model, and because Scania variability model contains around 50.000 features, it was outside the scope of the present paper to actually verify if these conditions are satisfied or not. However, although explicitly searched for, no verification artifact was identified that performs this kind of analysis. Therefore, a clear action for Scania is to start performing such analysis to ensure the absence of configuration mismatches. Currently, in line with common industrial practice (Mukelabai et al., 2018), the absence of configuration mismatches is partially verified during testing, because the identified verification documents contained testing results for several frequently ordered FLD variants. However, given that the total number of variants is around 24.000, the coverage with respect to the number of variants is rather low.

In summary, given the analyzed artifacts of the FLD system, and the conditions of Corollary 1, *it cannot be said* that the FLD system satisfies the contract $(\{\mathtt{A}_1\}, \mathtt{G}_1)$ in each possible configuration of a Scania vehicle. However, the identified inconsistencies between the specification and the implementation point to possible improvements of the FLD system-design. Moreover, the lack of exhaustive analysis for the absence of configuration mismatches, which is precisely formalized in Theorems 2 and 3, can be translated to an additional step in the engineering process that would increase the quality of Scania system. Therefore, as an answer to Q2, the use of the CBD framework can indeed lead to actionable engineering insights to improve the engineering process.

To set the answer to Q2 into a realistic context with respect to the required effort, note that using the CBD framework implies the need for a higher number of artifacts and traceability links, of higher quality. Compared to the practices observed in the case study this would certainly require the introduction of new processes and methods in the engineering process. Alternatively, construction of the CBD model may be performed on-demand from legacy artifacts, as shown in Section 6.3. The benefit of accepting the overhead may be seen from the observed mismatch between the FLD requirements and the corresponding implementation. This mismatch shows that *not using* a rigorous framework in design stages, must be compensated in implementation stages, but with a much higher effort. Detecting inconsistencies between the requirements in implementation stages means that implementation has been done with respect to inconsistent requirements, which must lead to the modification of requirements and implementation, which in turn must be verified again.

If the alternative is chosen, and CBD models are created on-demand from legacy artifacts, this implies a lesser but not an insignificant effort. In the case of the FLD system, the biggest effort was spent on the manual identification and collection of the relevant artifacts from different engineering tools and databases. This activity required about one person-week, but it should be noted that the relevant artifacts were in *data-silos* and in a more-digitalized toolchain this activity would be shorter. Given the

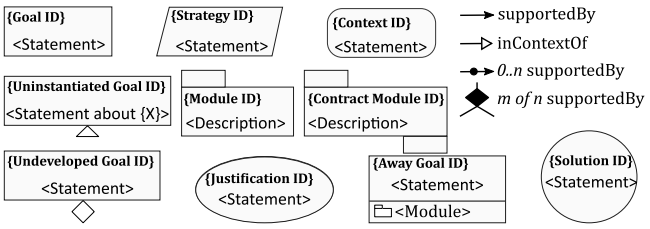


Fig. 9. GSN elements used in the present paper.

relevant artifacts, the CBD model was effectively created in two person-days. Because the unit of analysis included around one third of the total FLD system, creating the CBD model for the complete system would require additional effort. On the other hand, because the FLD system was already deployed, the reported effort corresponds to the creation of the model at the end of the development process. Since the benefits of the CBD framework materialize if used during the development, a more realistic scenario is that the CBD model is created iteratively, in parallel with the development process, thus reducing the overall effort.

7. An assurance case for a product line

This section presents the second contribution, which is a method to construct assurance-case arguments, based on a CBD model of a product line. The method allows the construction of *product-based* arguments (Habli and Kelly, 2006) that argue that a system satisfies a particular property in *all configurations*. Moreover, the method produces a *modular argument* in order to facilitate assurance-case maintenance, and to facilitate the creation of smaller pieces of argumentation in isolation, and their subsequent integration.

Because the CBD framework is notation independent, the following method is general. In this context, being general means that the method can be used to represent and reason about different dependability properties, e.g. safety, reliability, or security, for configurable systems of different size, and from different domains. This section relies on the GSN format to express assurance cases, but note that the GSN syntax has been mapped to *Structured-Assurance Case Meta-model* (SACM) (Object Management Group, 2019; Wei et al., 2019), which is an OMG standard that defines the *abstract syntax* for different *concrete syntaxes* that can be used to express an assurance case. Before defining the assurance-case construction method, the elements of the GSN syntax, which are relevant for the present paper, are introduced.

7.1. Goal-structuring notation

Besides the *Claims-Arguments-Evidence* (CAE) notation (Adelard, 1998), NOR-STA notation (Górski et al., 2012), and the concrete syntax for SACM (Object Management Group, 2019), the GSN notation (Origin Consulting (York) Limited, 2018) is the most prominent graphical syntax for expressing assurance cases. GSN is *"a graphical argument-notation which can be used to document explicitly the elements, and the structure of an argument and the argument's relationship to evidence"* (Origin Consulting (York) Limited, 2018). More specifically, GSN models form *rooted graphs* called *goal-structures*, which capture the assurance-case argumentation structure. Fig. 9 shows the GSN elements used in the present paper. A complete description of the GSN notation can be found in (Origin Consulting (York) Limited, 2018), and formalization of the goal-structures can be found in (Denney and Pai, 2018).

The elements in Fig. 9 have the following meaning: (i) *Goal* represents a *claim* about the system, (ii) *Strategy* represents the rationale for decomposing a goal into subgoals, (iii) *Solution* represents the evidence expected to support a claim within a goal, (iv) *Justification* explains why the rationale within a strategy is sound, (v) *Context* declares the context in which the claim should be interpreted, (vi) *Module* is a container to encapsulate smaller, self-contained goal-structures in order to facilitate the management of large assurance cases, (vii) *Away Goal* is a reference to a goal that is defined within a particular *Module*, and (vii) a *Contract Module* is a special type of module that defines how a goal within one module is supported by a goal from another module. Note that GSN contract modules are not directly related to the CBD design contract. The purpose of GSN contract modules is to *decouple* the argumentation between dependent GSN modules, while the purpose of CBD contracts is to *modularly* specify the behavior of a system.

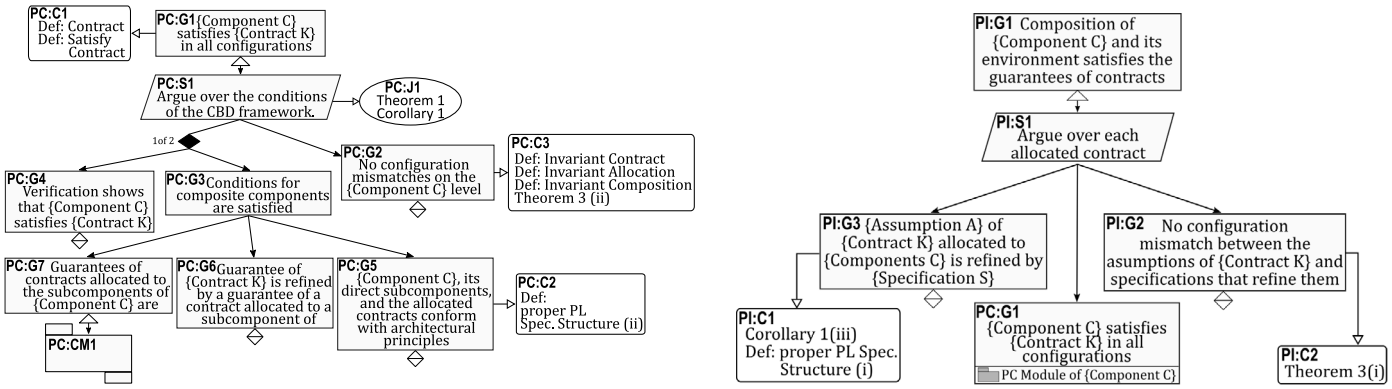
The described elements can be connected through the *supportedBy* and *inContextOf* links. The connections between the elements from Fig. 9 obey the following rules:

- permitted *supportedBy* connections are *goal-to-goal*, *goal-to-strategy*, *goal-to-solution*, *strategy-to-goal*, *goal-to-away goal*, *goal-to-module*, *goal-to-contract module*, *strategy-to-away goal*, *strategy-to-module*, and finally *strategy-to-contract module*
- permitted *inContextOf* connections are *goal-to-context*, *goal-to-justification*, *strategy-to-context*, *strategy-to-justification*, *goal-to-away goal*, *goal-to-module*, *strategy-to-away goal*, and finally *strategy-to-module*.

To allow specifying *template* goal-structures, i.e. assurance case *patterns* (Kelly and McDermid, 1997), additional GSN elements have been defined with the following meaning: (i) *Uninstantiated Goal* represents a claim that contains parameters which must be instantiated for a particular system, e.g. parameter X in Fig. 9, (ii) *Undeveloped Goal* represents a claim that must be further developed, i.e. connected to further goals, (iii) the *supportedBy* link with a black circle defines that the *multiplicity* of the *supportedBy* link is zero to many, and (iv) a black diamond splitting a *supportedBy* link into several ones is the so called *choice* element and it defines that *m* out of *n supportedBy* links are necessary for the argument to be complete.

7.2. Overview of the CBD-based pattern

This section presents the main idea for the assurance-case argumentation-creation method, namely a *modular assurance-case pattern*, which is based on the conditions of the CBD framework. More precisely, the method is based on two, mutually dependent patterns, which are shown in Fig. 10(b). The idea is that for each component *C*, the pattern in Fig. 10(a) will yield the argumentation that *C* satisfies the allocated contracts in all configurations. This captures the idea that each component can be developed in isolation with respect to its contract, i.e. with respect to the acceptable environments as defined by the declared assumptions (c.f. Definition 6). In addition, for all non-root components in the PL architecture, the pattern in Fig. 10(b) will yield the argumentation that within a particular system, i.e. represented by the root component, the assumptions of contracts are satisfied by other components comprising the system. In other words, once the different components are integrated into a system, the environment of each non-root component is fixed and it must be shown that this environment is indeed acceptable, i.e. it satisfies the declared assumptions. The reason why the pattern in Fig. 10(b) does not apply to the root component, i.e. to the overall system, is because the assumptions of the system are effectively



(a) Environment-independent pattern for each component C

(b) Pattern for the integration of component C with its environment

Fig. 10. Two dependent GSN patterns based on the CBD framework.

assumptions on the intended deployment environment of the system. For example, A_1 in Table 2 places an assumption on the driver of the vehicle.

Another important point is that the patterns induce a *modular architecture* in order to enable creation of smaller, self-contained arguments that capture conceptually different arguments. This means that for each component that instantiates the patterns from Fig. 10(b), the corresponding modules with the argumentation from the two patterns will be created. In this way, the overall argument is more robust with respect to possible assurance-case changes, but it also supports the allocation of responsibilities for particular modules to different engineering groups.

7.2.1. Description of the patterns

To distinguish between the elements of the patterns, we adopt the prefixes **PC** (*Pattern for Component*) and **PI** (*Pattern for Integration*) for element identifiers. Also note that some goals are both uninstantiated, and undeveloped. Here undeveloped means that once instantiated, these goals should be supported by evidence.

As indicated by goal **PC:G1**, the goal structure in Fig. 10(a) applies to any component C that has a contract K allocated to it. Recall that a component can represent any type of component, at any abstraction level, e.g. a complete product such as a vehicle, or a single software function. Strategy **PC:S1** states that the claim in **PC:G1** holds if the system represented by component C satisfies the conditions of the CBD framework. This strategy is asserted in the context of justification **PC:J1**, which references Theorem 1 and its product line extension in the form of Corollary 1. The remaining goals within the pattern claim that the conditions of the CBD framework are satisfied. Goal **PC:G2** claims that there are no configuration mismatches, as defined by the four invariance conditions in **PC:C3**. Note that Definition 16 and Theorem 3 (ii) apply *only to composite components* as they define invariant composition and absence of mismatches between the guarantee of a composite component and the guarantees of its direct subcomponents.

Goals **PC:G3** and **PC:G4** are mutually exclusive, as indicated by the *choice element* between them and the strategy element **PC:S1**. According to Corollary 1 (iv), goal **PC:G4** claims that verification shows that component C satisfies the allocated contract, if C is an atomic component. Goal **PC:G3** groups goals **PC:G5**-**PC:G7**, which are instantiated if C is a composite component. Goal **PC:G5** claims that C and its direct subcomponent conform with the *architectural principles*, which correspond to the conditions for a proper PL specification structure (c.f. Definition 12). Goal **PC:G6** claims that the guarantee of contract K is indeed refined, as required by Corollary 1 (iii), by some guarantee of a contract allocated to a subcomponent of C, as defined by Definition 9. Finally, goal **PC:G7** claims that the guarantees of contracts allocated to subcomponents of C are satisfied, i.e. the subcomponents of C behave as specified. Note that formally, the CBD framework defines the *satisfies* relation between a component and a specification while **PC:G7** does not precisely declare which component satisfies a specific guarantee. However, the idea is that from the perspective of component C, the subcomponents can be unknown, or can change occasionally. Thus, goal **PC:G7** is refined with respect to particular components and specifications within the contract module **PC:CM1**, which supports the goal **PC:G7**. In other words, contract module **PC:CM1** decouples the argumentation that component C satisfies its contracts, from the argumentation about its possibly unknown subcomponents.

As can be seen from Fig. 11, the contract module binds the module of component C, i.e. the goal **PC:G3**, with the modules obtained by instantiating the pattern from Fig. 10(b) for each subcomponent of C. When it comes to the pattern in Fig. 10(b), **PI:G1** claims that the composition of component C with its environment, i.e. with other components of the system, satisfies the guarantee of each contract allocated to C. The strategy in **PI:S1** states that the argument is over each contract allocated to component C. As discussed previously, for a guarantee of a contract to be satisfied, the assumptions of the contract must be satisfied by the environment, i.e. refined by a specification that is satisfied by another component of the system. This claim is captured by goal **PI:G3** and it corresponds to condition Definition 9 (i) which defines the specifications that can refine an assumption, and to condition Corollary 1 (iii) which requires that the refinement holds (c.f. **PI:C1**). For an assumption to be satisfied in all configurations, the refines relation between each assumption and the specification that refines it, must satisfy condition Theorem 3 (i) as captured by **PI:G2** and **PI:C2**. Finally, given an appropriate environment as claimed in **PI:G2** and **PI:G3**, the component C must satisfy the allocated contract in all configurations. Because this claim is the root claim of the pattern in Fig. 10(a), an *away goal* **PC:G1** refers to the root goal of pattern in Fig. 10(a). Note that instead of an away goal, a *contract module* could have been used. However, because the argument about the integration in a particular environment cannot be reused in any other environment, and also because the pattern in Fig. 10(a) can be instantiated only once for a particular component and corresponding contracts, decoupling the modules with a *contract module* would not bring significant benefits.

The fact that the patterns induce a modular architecture brings two benefits. Firstly, *monolithic* assurance cases are avoided and smaller pieces of argumentation can be created separately within particular models. Secondly, the assurance case modules, and the corresponding argumentation, can be created in parallel with the development process. For example, as the development of a component progresses, the pattern in Fig. 10(a) can be gradually

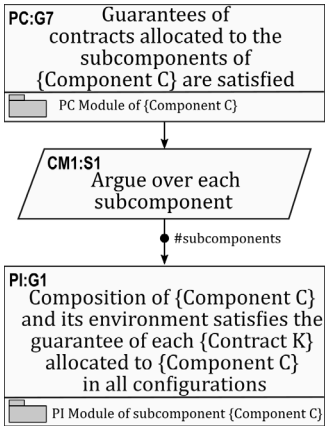


Fig. 11. Contents of the contract module **PC:CM1**.

instantiated and evidence created. Then, when the integration of the component starts, the pattern in Fig. 10(b) can be gradually instantiated and evidence created. In this way, the effort to create the assurance case is distributed across the development process and any given point in time, the focus is on particular modules of the assurance case, and not on its entirety.

Given the patterns from Fig. 10(b), the following section defines the instantiation procedure, and exemplifies the application of the pattern for the FLD system.

7.3. Pattern instantiation

In this section we define the procedure to instantiate the patterns from Fig. 10(b). The procedure is presented as a high-level pseudo-code to define the order in which different elements are instantiated, and also to define the correct CBD entities that can instantiate individual pattern elements. The instantiation procedure for the *environment independent* pattern from Fig. 10(a) comprises the following steps:

(ei1) Create a module for each component `C`, instantiate goal **PC:G1** for each allocated contract and create a single context element **PC:C1**,
(ei2) Create a GSN element **PC:S1** per instantiated goal **PC:G1**, and create a single **PC:J1**,
(ei3) Instantiate goal **PC:G2**, and create the context element **PC:C3**. Definition 16 and Theorem 3 (ii) should be a part of **PC:C3** only if component `C` is composite,
(ei4) If component `C` is atomic, instantiate goal **PC:G4**,
(ei5) If component `C` is composite, instantiate goals **PC:G5**-**PC:G7**, and create context **PC:C2**,
(ei6) Create solution elements that support the instantiated goals **P:G2**, and either **P:G4**, or **P:G5** and **P:G6**,
(ei7) If pattern from Fig. 10(b) has been instantiated for subcomponents of component `C`, instantiate contract module **PC:CM1**.

The instantiation procedure for the *environment dependent* pattern from Fig. 10(b) comprises the following steps:

(ed1) Create a module for each non-root component `C` and instantiate goal **PI:G1**,
(ed1) For each assumption of each contract allocated to `C`, instantiate goal **PI:G3** and create context **PI:C1**,
(ed2) For all contracts allocated to component `C`, instantiate goal **PI:G2** and create context **PI:C2**,
(ed3) Create solution elements that support the instantiated goals **PI:G3** and **PI:G2**,

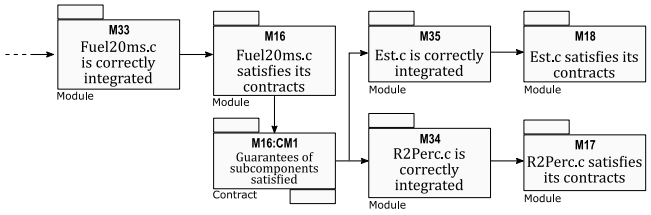


Fig. 12. Module view for a part of the assurance case for the FLD system.

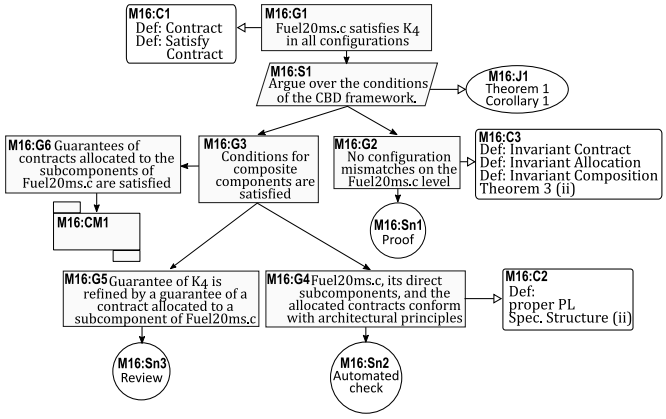


Fig. 13. Content of module **M16**.

(ed4) If the pattern from Fig. 10(a) has been instantiated for component `C`, instantiate the away goal **PC:G1**.

In conjunction with Fig. 10(b), steps s1–s7 and s1–s5 should be possible to convert into an algorithm that can be used to automate the creation of the assurance-case arguments. Such automation of assurance-case creation would go hand-in-hand with the long term vision to automate the creation and analysis of CBD models.

To exemplify the use of the instantiation procedure, Fig. 12 shows a part of the *modular architecture* for the assurance case for the FLD system. The complete modular architecture can be found in (Nešić). Fig. 12 shows two modules for the SW components `Fuel20 ms.c`, `Est.c`, and `R2Perc.c`, where the modules contain the argumentation obtained by instantiating the patterns from Fig. 10(b). According to the pattern in Fig. 10(a), module **M16** contains the argumentation that `Fuel20 ms.c` satisfies the allocated contract since, besides the claims in **M16**, the subcomponents of `Fuel20 ms.c`, namely `R2Perc.c` and `Est.c`, are integrated into their environment so that they satisfy the guarantees of their contracts. This argumentation is contained in modules **M34** and **M35**, where the contract module **M16:CM1** relates **M16** to **M34** and **M35**. According to the pattern in Fig. 10(b), besides the claims within **M34** and **M35**, these modules claim that `R2Perc.c` and `Est.c` satisfy the allocated contracts, but through away goals that belong to modules **M17** and **M18**.

To illustrate the content of the different modules, Fig. 13 shows the content of module **M16**, the content of module **M35** is shown in Fig. 14, Fig. 15 shows the content of module **M18**, Fig. 16 shows the content of the contract module **M16:CM1**, and the content of the remaining modules can be found in (Nešić).

The content of module **M16** in Fig. 13 is a straightforward instantiation of the uninstantiated goals with the component `Fuel20 ms.c` The content of module **M35** is more involved because the CBD model of the FLD system must be carefully considered. Goal **M35:G3** claims that, according to Definition 9 (i) and Corollary 1 (iii), A_6 is refined by G_7. An inspection of the

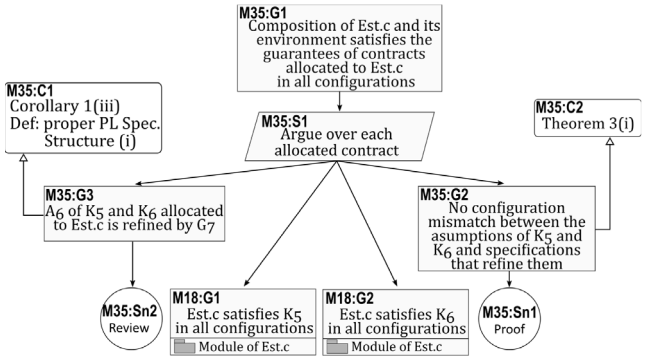


Fig. 14. Content of module **M35**.

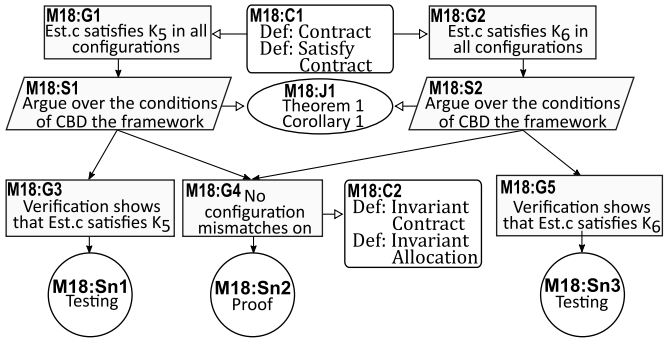


Fig. 15. Content of module **M18**.

CBD model in Fig. 8 shows that there exists an edge $e^{\mathtt{r}} = (\mathtt{G}_7, \mathtt{A}_6)$ which declares the intention that $\mathtt{G}_7$ should refine $\mathtt{A}_6$. Goal **M35:G2** claims that according to Theorem 3 (i), there is no configuration mismatch between $\mathtt{G}_7$ and $\mathtt{A}_6$. More precisely, it should hold that $p(\mathtt{G}_7) \models_{\mathcal{O}} p(\mathtt{A}_6)$. The claims in **M35:G2** and **M35:G3** ensure that the environment of `Est.c` satisfies the assumptions of the contract allocated to `Est.c` in all configuration. For the goal **M35:G1** to hold, it is also necessary to show that `Est.c` satisfies the allocated contracts, namely $(\{\mathtt{A}_6\}, \mathtt{G}_5)$ and $(\{\mathtt{A}_6\}, \mathtt{G}_6)$. These claims are captured by the away goals **M18:G1** and **M18:G2**.

Because module **M18** contains the argumentation that component `Est.c` satisfies two contracts, and because these two contracts share the assumption $\mathtt{A}_6$, the argumentation that the two contracts are satisfied by `Est.c` share goal **M18:G4**, context nodes **M18:C1** and **M18:C2**, and the justification node **M18:J1**. Besides these elements, and because `Est.c` is an atomic component, the module claims that verification shows that the two contracts are satisfied. In more general terms, this means that a single module contains the argumentation that a component satisfies each allocated contract. Note that because `Est.c` is an atomic component, module **M18** does not refer to further modules, i.e. **M18** is the leaf module in the modular architecture of the assurance case for the FLD system.

7.4. Relating CBD-based claims to other types of claims

As mentioned previously, the argumentation obtained by using the patterns from Fig. 10(b) must be integrated with other types of claims in order to obtain a complete assurance case. An advantage of the CBD-based argumentation is that it inherently supports connections to several other types of claims.

Firstly, and as noted in (Hutchesson and McDermid, 2013), certifying a product line must include the argumentation that

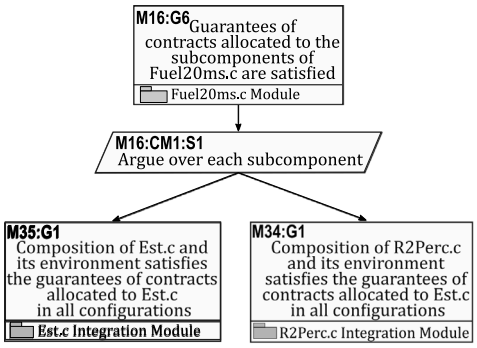


Fig. 16. Content of GSN contract-module **M16:CM1**.

the mechanisms for product line management are effective and reliable. More precisely, this means that it should be possible to determine with certainty whether all configurations that can be selected from a variability model are *valid*, whether a particular artifact applies to the intended configurations, and whether each variant (c.f. Definition 13) derived from the product line is valid. Each of these three types of claims can be developed from the CBD framework.

Throughout the present paper, it was assumed that only *valid* configurations can be selected from a variability model $\mathfrak{M}$ (c.f. Definition 1). Automated analysis of variability models has been investigated for decades (Benavides et al., 2010), where the analysis of whether a selected configuration is *valid*, is one of the most basic analysis. Moreover, it has also been shown that automated analysis of very large feature models is a tractable task (Mendonca et al., 2009). Another assumption throughout the present paper is that each presence condition φ is *consistent* with the variability model $\mathfrak{M}$, i.e. each presence condition defines a subset of the configurations defined by a variability model. Because each presence condition is effectively a partial configuration, the same techniques for verifying if a configuration is valid, can be used to verify if a presence condition is consistent with a variability model. Finally, Theorems 2 and 3, define the conditions under which it is guaranteed that each variant derived from a proper PL specification structure is a proper specification structure, which corresponds to a valid variant. The above argumentation can be expressed in the form of a GSN goal-structure to argue that product-line management mechanisms are effective and reliable.

The presented assurance case patterns yield the argumentation that the overall system, e.g. the FLD system, satisfies the allocated contract and that each constituent component, composed with other components of the system, satisfies the guarantees of the allocated contracts. This means that there is no module that contains the instantiation of the pattern in Fig. 10(b) for the overall system, e.g. the FLD component. As mentioned, the reason for this is that the assumptions of the system, e.g. assumption $\mathtt{A}_1$ of the FLD system, is effectively an assumption on the intended deployment environment, e.g. an assumption on the vehicle-driver. Although the pattern does not include the argumentation about the suitability of intended operating environment, because the assumptions of the system are explicitly declared, such argumentation could be developed, and connected to the argumentation based on the patterns in Fig. 10(b). This also means that if during deployment it is realized that the real-world environment exhibits new or different behavior than expected, the system assumptions will undergo changes and this will also have a traceable impact on the assurance case.

8. Discussion

In this section we answer research question RQ1 and RQ2 by considering the results of the exploratory case study, and by analyzing the assurance-case pattern from the point of view of the claimed benefits of the modular architecture, and from the point of view of feasibility to produce the required evidence.

8.1. Answering RQ1

The first research question asked if the CBD framework can be used to model arbitrary product lines and analyze if all variants satisfy a particular property without performing the analysis per-variant. Given the abstract nature of the CBD framework, the product-line extension of the CBD framework, and the results of applying the CBD framework to a real, industrial configurable system, there is a *strong indication* that the answer to RQ1 is positive, but a definitive answer would require further validation. Also, there are several notable facets to the answer to RQ1.

From the expressiveness perspective, we argue that the CBD framework is capable of representing an arbitrary product line. As was shown in the case study, the analyzed artifacts contained entities that could be directly interpreted as *components*, *specifications*, and *presence conditions*, and in general these concepts are widely used independently of the specific notation. Moreover, although not developed according to the CBD framework, the analyzed requirements documents already contained some contract-like requirements, which means that contract-like thinking is not uncommon. For example, components bought from suppliers were specified in terms of contracts. Traceability links that corresponds to *composition* and *allocation* relations were also explicitly declared in the analyzed documents. Finally, the *refines* relations were not explicit in the analyzed documents, but an attempt to make them explicit revealed a mismatch between the high-level design and system implementation.

On the other hand, in this particular case, the effort to use the CBD framework was not negligible. However, as discussed in Section 6.4.1, the reason for this were primarily the engineering practices and the quality of the available artifacts and traceability links for the analyzed system. Conceptually, the conditions of the CBD framework enforce rigorous development activities with detailed architectures, specifications, and traceability links. If this is not deemed as needed, then using the CBD framework might require too much effort compared to the possible benefits. However, for safety-critical system, rigorous development process is a requirement, therefore the conditions of the CBD framework provide valuable guidance.

This can be seen by inspecting the requirements of the functional-safety standard for the automotive domain, ISO 26262. Namely, as already discussed in Section 4, the concept of a *specification structure* already matches well with the idea of ISO 26262 where higher level requirements are broken down into lower level requirements, in parallel with the development of system components. When it comes to the management of configurable systems, ISO 26262 recognizes that an *item* can exist in multiple *variants*, and that the possible *variants* should be considered during system design, implementation, and verification. If considering all variants is not feasible, then considering a *reasonable, justified, and representative* subsets may be sufficient, where such subsets are primarily selected using *engineering judgment*. However, the standard does not offer guidance about what is *reasonable, justified*, or *representative*, while the concept of *engineering judgment* is in itself subjective. More detailed guidance exists for software where *configuration data*, which specifies the possible variants, must be *consistent* with the declared requirements, and it must be itself consistent. Furthermore, if there are *calibration variables* that impact the software run-time behavior, these must be consistent with the configuration data, with the declared requirements etc. However, the concept of consistency is not precisely defined.

The conditions of the extended CBD framework offer a technical solution for the above general concepts recognized by ISO 26262. Firstly, because the framework allows arguing about all variants, the need to identify a reasonable or a representative subset is avoided. Secondly, the concept of presence conditions can be used to capture the variability of all components, and not only software components. Thirdly, the invariance conditions from Definition 14- Definition 16 and from Theorem 3 directly correspond to the concept of *consistency* between the various types of artifacts that apply to multiple variants.

8.2. Answering RQ2

The second research question asked how a CBD model of a product line can be used to construct the corresponding assurance case. To answer RQ2, Section 7 presented two modular, assurance-case patterns, which define how a CBD model of product line can be used to construct the corresponding assurance case argumentation. However, because the pattern could have been defined differently to express the same argument, the following subsections discuss whether the modular architecture truly facilitates the stepwise creation and maintenance of an assurance case, and whether the required evidence is feasible to produce in a realistic engineering context.

8.2.1. Benefits of the modular architecture

To facilitate the management of assurance cases of realistic size, the pattern from Fig. 10(b) enforces a *modular architecture*. Firstly, having a coarse-grained representation of an assurance case in terms of modules, facilitates assurance-case maintenance and review on a high level. Secondly, as shown in Fig. 12, the modules are created for components at different abstraction levels, as defined for a particular system. In this sense, each module and the contained argumentation correspond to assurance claims and evidence originating from a particular development phase, which are possibly concurrent. At the same time, the usage of GSN contract modules allows the creation of modules independently, and then the modules can be related once the contract-modules are instantiated. Also note that because the modules capture the argumentation for particular components, the modules can be used to assign the responsibilities for their creation and maintenance. For example, module **M18** in Fig. 12 would probably be under the responsibility of a small development group, while modules **M16** and **M35** would be under the responsibility of the group responsible for the `COO SW`. Finally, the separation into two patterns, contained by two separate modules, directly supports the idea of *safety element out of context* (SEooC), as defined by ISO 26262. The purpose of SEooC is to allow suppliers to develop safety-critical components with respect to a contract provided by their customers, while the supplier's customers are responsible for the correct integration of SEooC components into their systems. In summary, despite the need for further validation, the current modular architecture suggests that the creation of the argumentation within the modules can be tightly integrated with a typical development process.

Although widely considered beneficial, an industrial evaluation (Kelly and Bates, 2005) reports that there are two major challenges when using modular argumentation: (i) deciding on the content of modules such that references between the modules, the so-called *module interface*, are clearly defined, and (ii) ensuring that the composition of all modules yields a valid argument because the assumptions, context, and evidence between different modules might be incompatible. The present

paper overcomes these two challenges by relying on the CBD framework. Firstly, each *supportedBy* relation between two modules, as defined by corresponding contract-modules, corresponds to the composition of components with respect to the guarantees relevant for a particular instantiation of pattern in Fig. 10(b). Regarding the second challenge, the modularization in Fig. 12 is simply a distribution of arguments about premises of Corollary 1 across multiple modules, but the overall argument is preserved and thus also the argument validity.

8.2.2. Feasibility of producing the required evidence

This section considers each type of evidence required by the assurance-case pattern from Fig. 10(b) and discusses the state-of-the-art techniques to produce such evidence, and the state-of-practice in Scania. In other words, given that each evidence corresponds to an analysis of the constructed PL specification structure with respect to the conditions of Corollary 1, this section discusses how to practically analyze a PL specification structure.

Absence of configuration mismatches. Evidence such as **M16:Sn1**, **M35:Sn1** and **M18:Sn2** is produced by analyzing the constructed PL specification structure against conditions (i) and (ii) of Corollary 1. The work in (Nešić and Nyberg, 2018) shows how such evidence can be produced using *description logic*. In general, such evidence can be routinely performed by using basic SMT solving (De Moura and Bjørner, 2008). For example, in the context of the model in Fig. 8, verifying if the allocation of $(\mathtt{A}_5, \mathtt{G}_4)$ to component $\mathtt{C}_{\mathtt{Fuel20ms}}$ is *invariant* (c.f. Definition 15), means verifying if $p(\mathtt{G}_4) \models_{\Theta} p(\mathtt{C}_{\mathtt{Fuel20ms}})$, i.e. $\varphi_{27} \models_{\Theta} \varphi_7$. Practically, this implies building an SMT model that contains each $\theta_i \in \Theta$ from a variability model, the assertion $\neg(\varphi_{27} \rightarrow \varphi_7)$, and checking if such SMT model is satisfiable. If the model is satisfiable, then the allocation is not *invariant*, and vice versa. Note that although the Θ of a large variability model contains many constraints and consequently results in a large SMT model, once each $\theta_i \in \Theta$ is asserted, verifying different invariance constraints requires changing a single expression, namely the negated implication between presence conditions, i.e. $\neg(\varphi_j \rightarrow \varphi_k)$. Also, as shown in (Mendonca et al., 2009), this type of satisfiability checks is tractable in practice. In summary, well-known, scalable techniques to produce this type of evidence exist, but as discussed in Section 6.4.2, they are not adopted in Scania.

Conformance with architectural principles. Producing the evidence such as **M16:Sn2** in Fig. 13, which corresponds to the analysis of the constructed PL specification structure against the conditions of Definition 12, can be more or less challenging. Because the design and implementation happen in multiple phases that are supported by different tools, and performed by different engineering groups, ensuring that the same principles are uniformly followed is challenging. If it is possible to extract, merge, and possibly transform the relevant artifacts into a formal model, then this evidence can be produced automatically. Under such assumptions, the work in (Nešić and Nyberg, 2018) presents a method based on *description logic* to produce such evidence in an automated way. Additional tools that could probably be used to perform such analysis are *Alloy* (Jackson, 2002) or *Clafer* (Bąk et al., 2016).

Previous approaches for single system assurance (Diskin et al., 2018; Hawkins et al., 2015; Habli et al., 2010) frame this type of problem as the central activity in *Model-Based Engineering* (MBE) paradigm, namely the instantiation of a *meta-model* into a *model*. While such approaches also promise high levels of automation, they may not be always applicable because the engineering context might not be fully MBE compliant. In the example of the FLD product line, the logical architecture of the FLD system was represented by a well-defined model (c.f. Table 1), and the corresponding tool performed consistency checks, but there was no model of the software architecture.

Specification refinement. Regarding the evidence about *specification refinement*, e.g. *solution* elements **M16:Sn3** and **M35:Sn2** in Fig. 14, this corresponds to the analysis of the constructed PL specification structure against condition (iii) of Corollary 1. In general, this is a hard problem for any method. Usually, it is more likely that implementation-level specifications, e.g. the ones allocated to software or hardware components, are susceptible to the application of formal methods that can prove refinement because they are typically more detailed. The work in (Filipovikj et al., 2018) introduces an automated, SMT-based method to verify specification refinement, and exemplifies the approach also on the FLD system. However, it is a widely acknowledged fact that writing formal requirements is a difficult and tedious task, thus in a realistic industrial context, *expert review* will probably be the verification method of choice. It should be noted that among the analyzed documents of the FLD product line, there was no explicit evidence of specification refinement, but each requirement document had to be analyzed and approved by a senior engineer. Thus, this type of evidence is probably implicitly created, although without documenting the reasons to approve or reject a requirement document.

Component behavior. The fourth and final type of evidence, e.g. *solution* elements **M18:Sn1** and **M18:Sn3** in Fig. 15, is about showing that each contract *allocated to* an atomic component is *satisfied*, and this corresponds to analyzing the PL specification structure against condition (iv) of Corollary 1. As with other analyses, depending on the level of formality, this analysis can be more or less automated. In the case of software components where the assumptions and guarantees are expressed in a formal language, formal software-verification approaches against contract-based specifications can often produce good results (Gurov et al., 2017). On the other hand, the most common, and a *de-facto* standard method in industry, to verify component behavior is *unit* and *integration* testing. The analyzed FLD documentation confirms this as it contained reports of extensive testing. Note that the argumentation based on the CBD framework requires only the verification of atomic component, i.e. unit testing, but because the complexity of systems can lead to unforeseen interactions between components, integration testing id often used as a method to ensure absence of unwanted interactions (Mariani et al., 2007). For example, goal **M35:G1** in Fig. 14 could be supported by an *additional argumentation leg*, that claims that no unwanted interactions were observed, supported by integration-testing results.

9. Related work

Section 1.1 has already discussed two approaches that are directly related to the presented approach. However, there are other approaches that tackle the challenge of representing a product line, or the challenge of argumentation reuse, potentially in assurance cases for product lines.

9.1. Frameworks to represent a product line

One line of research uses the ideas of *model-based engineering* to represent a product line, typically as a series of UML (Ziadi and Jezequel, 2006; Gomaa, 2011; Junior et al., 2010) or SysML (Chiquitto et al., 2015; Gaeta and Czarnecki, 2015) models by defining *stereotypes* to support product line engineering. Both UML and SysML are comprehensive, semi-formal modeling frameworks that support a wide range of concrete artifacts and traceability links, definitively a greater number than the CBD framework. However, unlike the CBD framework, UML and SysML do not support *compositional reasoning*, and this the fundamental reason why any analysis must be performed *per-variant*. Furthermore,

although UML and SysML are mature frameworks for everyday engineering, assuming a representation of a product line in a concrete engineering framework such as UML or SysML would implicitly place requirements on the development methods and process, and a major part of the motivation to use the CBD framework is exactly to avoid doing so.

Another line of research comes from the domain computer science (Classen et al., 2012; Cordy et al., 2013; Asirelli et al., 2011), where various types of *labeled transition systems* are used to model system variants in terms of their states, and transitions between the states, where the transitions are labeled with presence conditions. Then, formal specifications, expressed in various types of logic, are verified over the transition systems by using model checking techniques. Due to their abstract and formal nature, such frameworks can be used to represent and rigorously analyze arbitrary systems. Moreover, compositional reasoning is well-defined for transition systems, i.e. it is possible to compose transition systems and deduce the properties of the resulting composition. However, on a practical level, such approaches abstract away the traceability links between the artifacts such as *allocated to*, or *intended refinement* from the CBD framework. Moreover, because of their fully formal nature, it is unlikely that a significant number of components will be represented as transitions systems and that a significant number of specifications will be expressed as logical formulas instead of the typically informal and semi-formal formats (Woodcock et al., 2009). Finally, as is often the case, this level of formality is achieved by trading expressiveness, i.e. concrete logical languages allow expressing a restricted class of specifications, and transition systems allow the modeling only of discrete systems, while the CBD framework makes no such assumptions.

9.2. Argumentation reuse

Two schools of thought exist for the construction of argumentation structures. The first line of research has focused on capturing *assurance case patterns* (Denney and Pai, 2013) that have been used in real world assurance cases or in case studies performed by researchers. The second line of research has focused on the creation of assurance-case arguments by analyzing various types of artifacts. In the context of product lines, besides the approaches described in the introduction (Habli, 2009; Habli and Kelly, 2010; de Oliveira et al., 2015; Hutchesson and McDermid, 2010, 2011, 2013), most notable contributions come from the latter group (Gallina et al., 2013; Sljivo et al., 2017, 2016; de la Vara et al., 2019; Gallina et al., 2014; Javed and Gallina, 2018).

The work in (Gallina et al., 2013) presents a method called *VROOM and cC*, which focuses on achieving compliance of a product line with the first two phases of the ISO 26262 standard in the automotive domain. The main idea of the approach is to align each step of the safety lifecycle proposed by ISO 26262, with the product-line engineering idea of modeling *common* and *variable* parts of engineering assets within a product line. The idea leads to a *double, V development model*, where in each phase of the model, there is a set of recommended, tool-supported, engineering activities which should enable systematic, fine-grained, tracing of engineering assets within a product line. Compliance with the first two phases of ISO 26262 is shown by leveraging the established traceability to create an assurance case which reflects the variability within the engineering assets. In other words, this approach follows the same idea as the work in (Habli, 2009; Habli and Kelly, 2010; de Oliveira et al., 2015), where the assurance case arguments are for all system variants are *syntactically integrated*, but it is assumed that per-variant arguments are *derived* and analyzed if they are *valid and convincing*.

The line of work in (Sljivo et al., 2017, 2016) provides an approach for the creation of reusable assurance cases fragments based on a semi-formal notion of *weak* and *strong safety contracts*. The approach considers the concept of *Safety Element out of Context* (SEooC) from the standard ISO 26262, which is used to represent the components that OEMs procure from external suppliers. The essence of the approach is to perform hazard analysis and risk assessment for a SEooC using a variant of *Fault Propagation and Transformation Calculus* (FPTC) (Gallina et al., 2012) and then transform the analysis results into safety contracts. The present paper considers development and assurance of arbitrary product lines, and assurance of arbitrary dependability properties, from the OEM perspective, and because of that relies on a more expressive contract-based framework which subsumes the *weak* and *strong* safety contracts. Consequently, the approach in the present paper argues about system-level dependability properties instead of only safety of SEooC. Furthermore, unlike the approach in (Sljivo et al., 2016), where presence conditions are contract assumptions, presence conditions in the present paper are orthogonal to components and specifications. This allows presence conditions and the corresponding artifacts to evolve separately.

Unlike the work in the present paper, which focuses on *product-based arguments*, the line of work in (Gallina et al., 2014; Javed and Gallina, 2018) focuses explicitly on *process-based* arguments that can be *reused*. The reuse can occur between different yet similar systems, and in that sense this work is applicable to product-line engineering. However, reuse of process arguments can also occur between different domains, regulated by different standards (Gallina et al., 2014). The central artifact in this line of work is a model of the used engineering processes, with variable process steps, expressed in the SPEM format (Object Management Group, 2008). The variability of the process is modeled either with the build-in capabilities of SPEM format (Gallina et al., 2014), or using the integration between the BVR language (Haugen and Øgård, 2014) and the implementation of SPEM within the EPF composer[1]. Once the process model is in place, the so-called *process line*, variants of the process can be *derived*, and process-based arguments can be constructed for different process variants. Moreover, the approach can be also used to detect some types of *fallacies* within process-based arguments (Muram et al., 2018).

Finally, it should be noted that the work from (Gallina et al., 2014; Javed and Gallina, 2018; Sljivo et al., 2017) has been integrated into a tool ecosystem AMASS (de la Vara et al., 2019), which support additional methods for analysis and assurance of arbitrary dependability properties of single systems. For example, the work in (Javed et al., 2019) exploits the AMASS tool ecosystem to show how impact analysis can be performed between a product-line and a process-line specification in the BVR and the EPF tool.

10. Threats to validity

Construct validity. The answers to research question depend on the results of the exploratory case study. To avoid the risk of analyzing the wrong artifacts, e.g. mismatching versions, or wrongly interpreting the artifacts, e.g. the real intention of a requirement, domain experts were included in all phases of the study before starting the CBD model synthesis. Because the CBD concepts such as *component*, *specification*, *allocated to* relation etc., are generic, and defined so that they match closely to engineering practices, the risk of misinterpreting an artifact as a CBD concept was low. For example, the C-language implementation contains C-functions, which were mapped to components $C_{\texttt{R2Perc}}$ and $C_{\texttt{Est}}$, a *.c file of the software-application component is mapped to $C_{\texttt{Fuel20ms}}$, the analyzed version of the COO source code is mapped to $C_{\texttt{COOSW}}$ etc. Similarly, each considered requirement from requirements documents was directly mapped to a single specification *S*.

[1] https://www.eclipse.org/epf/

Internal validity. To minimize the risk of using an unsuitable framework for product-line modeling and analysis, Section 9.1 compares the CBD framework to other feasible frameworks. When the initial mapping of the analyzed artifacts did not yield a proper PL specification structure, it was decided to *infer* the missing information instead of using a method to elicit the information from the domain experts. In this way, we avoided any bias that domain experts could have introduced, and also avoided the risk that the domain experts misinterpret the conditions of the CBD framework. Given that the authors defined the inference rules, and to avoid any bias during the inference of the missing information, we took care that no new knowledge is introduced. Because of that, as can be seen in Table 2, each inferred specification is always equivalent to an explicitly declared Scania requirement. Moreover, the inference process was guided by explicitly, and formally defined conditions of the CBD framework, which further minimizes the risk of introducing bias. It should be noted that during the inference process, a slightly different set of inferences was possible. For example, Table 2 states that $\mathtt{G}_1 \equiv \mathtt{G}_2 \equiv \mathtt{G}_3$, where $\mathtt{G}_2$ and $\mathtt{G}_3$ were inferred to satisfy Definitions 9 and (ii), and Corollary 1 (iii), although the edge $e^r = (\mathtt{G}_4, \mathtt{G}_3)$ violates Corollary 1 (iii). Alternatively, it could have been inferred that $\mathtt{G}_2 \equiv \mathtt{G}_3 \equiv \mathtt{G}_4$, which also leads to a *proper* PL specification structure, but then edge $e^r = (\mathtt{G}_2, \mathtt{G}_1)$ would violate Corollary 1 (iii). In other words, because no new knowledge is added, and the same inference rules are used, different alternatives would still lead to a *proper* PL specification structure, and to the violation of the same semantic conditions, e.g. the ones in Corollary 1.

External validity. The system selected for the case study, namely the FLD system, is a typical automotive system. It is a mature, currently *in-production* system with thousands of unique system-variants. The system includes various types of components, e.g. software, hardware, and mechanical, and it is also safety-critical which suggests that an assurance case should be developed for it. Moreover, the FLD system proved to be a particularly challenging system for an application of the CBD framework because the used engineering practices were not aligned with the practices recommended by the relevant safety standard, and which are directly supported by the CBD framework. However, despite the challenges, it was possible to apply the CBD framework. These factors strongly suggest that the extended CBD framework is able to model and analyze *arbitrary and realistic* product lines.

Reliability. To enable the reproducibility of the exploratory study, and the applicability of the inference steps, the applied methodology was described in detail in Section 6.1. Moreover, for each entity and traceability link that was inferred, there is an explicit description about how that was achieved. One limitation to the reliability of the study is the inability share the raw Scania artifacts due to confidentiality issues. This limitation was partly alleviated by providing an online repository with assurance-case related artifact such as the assurance-case pattern, the complete modular architecture for the FLD system etc.

11. Conclusion

To satisfy the needs of as many customers as possible, enterprises are increasing the allowed levels of customization, even for complex *cyber–physical systems* that must satisfy critical dependability properties such as safety, security or reliability. Making such systems highly configurable, e.g. engineering them as *product lines*, hinders the use of existing, single-system assurance methods because analyzing and assuring individual system configurations becomes infeasible.

This paper has introduced a novel and general approach for the creation of an assurance case for a *complete product line*, instead of for each individual system configurations. The first contribution of the paper is a product-line extension of a general-purpose *contract-based design-framework*, in order to obtain a rigorous, holistic model of all system configurations within a product line. The result of this extension is Corollary 1, which summarizes the conditions under which it can be deduced that all system configurations satisfy a particular critical property. The applicability of the extended framework is evaluated through an exploratory case study on a part of a real product-line from Scania. The second contribution defines a *modular, assurance-case pattern*, which is based on the conditions from Corollary 1, and which can be instantiated to create *product-based* argumentation about a complete product line. A comparison to the requirements of the ISO 26262 standard, as well with observed practices in Scania shows that the usage of the pattern can be aligned with a typical engineering process. Also, the majority of the required evidence is either already typically produced, or it can be produced with well-known automated techniques.

CRediT authorship contribution statement

Damir Nešić: Conceptualization, Methodology, Formal analysis, Writing - original draft, Data curation. **Mattias Nyberg:** Validation, Writing - review & editing, Supervision. **Barbara Gallina:** Writing - review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

D. Nešić and M. Nyberg were supported by the ECSEL PRYSTINE project (No 783190) and by Scania CV AB. B. Gallina was partly supported by ECSEL AMASS project (No 692474).

Appendix. Theorem proofs

This section presents the proofs of Theorems 1–3.

Theorem 1. *Let $\mathscr{A}$ be an architecture with the root C_r, and let $\mathcal{K}$ be a set of contracts such that each $K \in \mathcal{K}$ is allocated to at least one component from $\mathscr{A}$ and contract $K_r \in \mathcal{K}$ is allocated to C_r. Let $\mathcal{K}_{At} \subseteq \mathcal{K}$ be the set of contracts allocated to atomic components $\mathcal{C}_{At} \subseteq \mathscr{A}$. If*

(i) $\forall K \in \mathcal{K}_{At}.\forall C \in \mathcal{C}_{At}.\mathtt{allTo}(K, C) \rightarrow C \blacktriangleright K$,
(ii) $\mathcal{K}$ forms a proper specification structure $\mathfrak{D}$,
(iii) for each edge $e^r = (S_1, S_2)$, it holds that $S_1 \sqsubseteq S_2$,

then it holds that $C_r \blacktriangleright K_r$. □

Proof. To avoid notational overhead, we present the proof for a *two-level* architecture $\mathscr{A}$ with a root component C_r whose subcomponents are atomic components. The proof for an architecture $\mathscr{A}$ with an *arbitrary number* of levels corresponds to applying the following reasoning to each composite component.

Let contract K_r be such that it contains n assumptions. According to Definition 7, we can rewrite the theorem claim as $\forall C_e.C_e \rhd (A_1^r \sqcap \cdots A_n^r) \rightarrow C_e \otimes C_r \rhd G_r$. Given that the theorem claim is an implication, we prove it by using a direct-argument strategy.

Assume an arbitrary C_e such that $C_e \rhd (A_1^r \sqcap \cdots A_n^r)$. From premise (ii) and according to Definition 9–(ii), there exists a contract $(\mathcal{A}_n, G_n)$, allocated to a component $C_m \in \mathcal{C}_{At}$, such the there exists an edge $e^r = (G_n, G_r)$. Because $\mathcal{A}_n$ is possibly empty, there are two cases. In case a), $\mathcal{A}_n = \emptyset$ and from this and premise (i) it follows that $C_m \rhd G_n$. From this and premise (iii), it follows that $C_m \rhd G_r$. Because of monotonicity, C_m can be composed with C_e and components from $\mathcal{C}_{At} \setminus C_m$ that comprise C_r, i.e. $C_e \otimes C_r \rhd G_r \leftrightarrow C_r \blacktriangleright K_r$. In case b), $\mathcal{A}_n \neq \emptyset$, and from premise (ii), according to Definition 9–(i), it follows that for each $A_j^n \in \mathcal{A}_n$ there exists an edge $e^r = (S, A_j^n)$ where $S \equiv A_i^r$, or $S \equiv G_k$ where G_k is a guarantee of a contract $(\mathcal{A}_k, G_k)$ allocated to a component C_p, which is a sibling of C_m. If $S \equiv A_i^r$, given assumption $C_e \rhd (A_1^r \sqcap \cdots A_n^r)$ and premise (iii), it follows that $C_e \rhd A_j^n$ and from premise (i) it follows that $C_e \otimes C_m \rhd G_n \leftrightarrow C_e \otimes C_m \rhd G_r$. Similarly to case (i), due to monotonicity, $C_e \otimes C_m$ can be extended to $C_e \otimes C_r \rhd G_r$. If $S \equiv G_k$, where G_k is a guarantee of a contract $(\mathcal{A}_k, G_k)$, the previous reasoning process can be repeated for $\mathcal{A}_k$.

The previous steps can be repeated for each set of assumptions, such as $\mathcal{A}_n$, that are ancestors of G_r in the graph of $\mathfrak{D}$. Because the graph of $\mathfrak{D}$ is finite and does not contain directed cycles, exploring all such assumptions must terminate and result in an expression $C_x \otimes \cdots \otimes C_m \rhd G_n$. This expression either already contains C_e, or due to monotonicity it can be expanded with C_e and any components that comprise C_r but are not included in the expression. From this it follows that $C_e \otimes C_r \rhd G_n$, and given premise (iii) if follows that $C_e \otimes C_r \rhd G_r \leftrightarrow C_r \blacktriangleright K_r$, which concludes the proof.

Theorem 2. *Let Γ be a set of valid configurations, and let $\widehat{\mathfrak{D}}$ be a proper PL specification structure. If*

(i) each contract is invariant,
(ii) allocation of contracts to components from $\widehat{\mathscr{A}}$ is invariant,
(iii) composition of child into parent components in $\widehat{\mathscr{A}}$ is invariant,

then each variant $\mathfrak{D}_\gamma$, is a *specification structure.* □

Proof. Let $\gamma \in \Gamma$ be an arbitrary configuration, let $\mathfrak{D}_\gamma$ be the corresponding instantiated PL specification structure, and let $\varphi \in \Phi_{\widehat{\mathfrak{D}}}$ be a presence condition such that $eval(\varphi, \gamma) = true$. If φ is the presence condition of some guarantee G_j, i.e. $p(G_j) = \varphi$, then from premise (i) (c.f. Definition 14–(ii)) it follows that each of the corresponding assumptions $A_i^j \in \mathcal{A}_j$ is such that $eval(p(A_i^j), \gamma) = true$ and the pair $(\mathcal{A}_j, G_j)$ is a contract. From this it follows that for each $A_i^j \in \mathcal{A}_j$ there exists a node $n_i \in \mathcal{N}_\gamma$ that represents A_i^j, and there exists a node $n_{j \neq i} \in \mathcal{N}_\gamma$ that represents G_j. Also, it follows that there exist edges $e^a = (A_i^j, G_j)$ for each $A_i^j \in \mathcal{A}_j$. By using similar reasoning, and from premise (i) (c.f. Definition 14–(i)), identical conclusions follow if φ is the presence condition of some assumption A_i^j. Thus, given premises (i), and because according to Definition 11 the graph of $\widehat{\mathfrak{D}}$ conforms with the conditions from Definition 8, it follows that the instantiated PL specification structure $\mathfrak{D}_\gamma$ forms a directed graph according to Definition 8.

Definition 8 also requires that each contract from $\mathfrak{D}_\gamma$ is allocated to a at least one component from $\mathscr{A}_\gamma$, and that $\mathscr{A}_\gamma$ is an architecture according to Definition 4. Regarding contract allocation, from premise (ii), for each guarantee G_j of a contract $(\mathcal{A}_j, G_j)$ such that $eval(p(G_j), \gamma) = true$, it follows that there exists a component $C_c \in \mathscr{A}_\gamma$ such that $\texttt{allTo}((\mathcal{A}_j, G_j), C_c)$, and $eval(p(C_c), \gamma) = true$. From this, and regarding $\mathscr{A}_\gamma$ being an architecture according to Definition 4, if $C_c \in \mathscr{A}_\gamma$ is the root of $\widehat{\mathscr{A}}$, and because $\mathscr{A}_\gamma$ is a tree (c.f. Definition 13–(iv)), it follows that $\mathscr{A}_\gamma$ is an architecture according to Definition 4 for the configuration γ. If C_c is a non-root component from $\widehat{\mathscr{A}}$, then from premise (iii) it follows that there exists a component C_p, which is the *parent* of C_c in $\mathscr{A}_\gamma$, such that $eval(p(C_p), \gamma) = true$. Similarly, if C_p is also a non-root component in $\widehat{\mathscr{A}}$, then from premise (iii) the presence condition of the parent of C_p also evaluates to *true*. By repeating this process, it follows that the presence condition of component C_r, which is the root of $\widehat{\mathscr{A}}$, evaluates to true and it holds that $C_r \in \mathscr{A}_\gamma$. From this, and because $\mathscr{A}_\gamma$ is a tree (c.f. Definition 13–(iv)), it follows that $\mathscr{A}_\gamma$ is an architecture according to Definition 4 for configuration γ. This concludes the proof.

Theorem 3. *Let Γ be a set of valid configurations, and let $\widehat{\mathfrak{D}}$ be a proper PL specification structure such that each variant $\mathfrak{D}_\gamma$ is a specification structure. If*

(i) for each $A_i^j \in \mathcal{S}$, and edges $e^r = (S_1, A_i^j), \ldots, (S_n, A_i^j)$ it holds that $p(A_i^j) \models_\Theta \bigvee_{k=1}^n p(S_k)$,
(ii) for each $G_j \in \mathcal{S}$, and edges $e^r = (G_1, G_j), \ldots, (G_n, G_j)$ it holds that $p(G_j) \models_\Theta \bigvee_{k=1}^n p(G_k)$,

then each $\mathfrak{D}_\gamma$ is proper *specification structure.* □

Proof. The proof consists of showing that conditions from Definition 9 hold for each instantiation of $\widehat{\mathfrak{D}}$. Let $\gamma \in \Gamma$ be an arbitrary configuration and let $\mathfrak{D}_\gamma$ be an instantiation of $\widehat{\mathfrak{D}}$ for configuration γ. From premise (i) it follows that the instantiation $\mathfrak{D}_\gamma$ is such that for each A_i^j, where $eval(p(A_i^j), \gamma) = true$, there exists a specification S_k such that $eval(p(S_k), \gamma) = true$. From this, and because $\mathcal{E}_\gamma \subseteq \mathcal{E}$ (c.f. Definition 13–(ii)) the edge $e^r = (S_k, A_i^j)$ conforms to the condition (i) of Definition 9. In other words, condition (i) from Definition 9 holds for $\mathfrak{D}_\gamma$.

The fact that condition (ii) from Definition 9 holds for $\mathfrak{D}_\gamma$ follows directly from premise (ii). Finally, according to premise (iii), $\mathfrak{D}_\gamma$ does not contain directed cycles. Consequently condition (iii) of Definition 9 holds. Given that γ is arbitrary, the above proof holds for each instantiation of $\widehat{\mathfrak{D}}$.

References

Adelard, L.L.P., 1998. ASCAD: Adelard Safety Case Development Manual. Technical Report, CAE.

Apel, S., Batory, D., Kästner, C., Saake, G., 2016. Feature-Oriented Software Product Lines. Springer, http://dx.doi.org/10.1007/978-3-642-37521-7.

Asirelli, P., ter Beek, M.H., Gnesi, S., Fantechi, A., 2011. Formal description of variability in product families. In: International Software Product Line Conference. SPLC, pp. 130–139. http://dx.doi.org/10.1109/SPLC.2011.34.

Bąk, K., Diskin, Z., Antkiewicz, M., Czarnecki, K., Wąsowski, A., 2016. Clafer: unifying class and feature modeling. Softw. Syst. Model. 15 (3), 811–845.

Benavides, D., Segura, S., Ruiz-Cortés, A., 2010. Automated analysis of feature models 20 years later: A literature review. Inf. Syst. 35 (6), 615–636. http://dx.doi.org/10.1016/j.is.2010.01.001.

Bensalem, S., Bozga, M., Sifakis, J., Nguyen, T.-H., 2008. Compositional verification for component-based systems and application. In: International Symposium on Automated Technology for Verification and Analysis. Springer, pp. 64–79. http://dx.doi.org/10.1007/978-3-540-88387-6_7.

Benveniste, A., Caillaud, B., Nickovic, D., et al., 2018. Contracts for system design. Found. Trends Electron. Des. Autom. 12, 124–400. http://dx.doi.org/10.1561/1000000053.

Chapman, R., Schanda, F., 2014. Are we there yet? 20 years of industrial theorem proving with SPARK. In: Interactive Theorem Proving. ITP, Springer, pp. 17–26. http://dx.doi.org/10.1007/978-3-319-08970-6_2.

Chiquitto, A., Gimenes, I.M.S., Oliveira, E., 2015. SyMPLES-CVL: A SysML and CVL based approach for product-line development of embedded systems. In: Brazilian Symposium on Components, Architectures and Reuse Software. pp. 21–30. http://dx.doi.org/10.1109/SBCARS.2015.13.

Cimatti, A., Tonetta, S., 2012. A property-based proof system for contract-based design. In: Euromicro Conference on Software Engineering and Advanced Applications. SEAA, IEEE, pp. 21–28. http://dx.doi.org/10.1109/SEAA.2012.68.

Classen, A., Cordy, M., Schobbens, P.-Y., Heymans, P., Legay, A., Raskin, J.-F., 2012. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. IEEE Trans. Softw. Eng. 39 (8), 1069–1089. http://dx.doi.org/10.1109/TSE.2012.86.

Clements, P., Northrop, L., 2001. Software Product Lines: Practices and Patterns. Addison-Wesley Professional.

Cordy, M., Schobbens, P.-Y., Heymans, P., Legay, A., 2013. Beyond boolean product-line model checking: dealing with feature attributes and multi-features. In: International Conference on Software Engineering. ICSE, IEEE, pp. 472–481.

Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K., Wąsowski, A., 2012. Cool features and tough decisions: a comparison of variability modeling approaches. In: International Workshop on Variability Modeling of Software-Intensive Systems. VAMOS, ACM, pp. 173–182. http://dx.doi.org/10.1145/2110147.2110167.

De Moura, L., Bjørner, N., 2008. Z3: An efficient SMT solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. TACAS, Springer, pp. 337–340. http://dx.doi.org/10.1007/978-3-540-78800-3_24.

De Roever, W.-P., Langmaack, H., Pnueli, A., 2003. Compositionality: The Significant Difference: International Symposium. COMPOS'97, Springer, http://dx.doi.org/10.1007/3-540-49213-5.

Denney, E., Pai, G., 2013. A formal basis for safety case patterns. In: Computer Safety, Reliability, and Security. SAFECOMP, Springer, pp. 21–32. http://dx.doi.org/10.1007/978-3-642-40793-2_3.

Denney, E., Pai, G., 2018. Tool support for assurance case development. Autom. Softw. Eng. 25 (3), 435–499. http://dx.doi.org/10.1007/s10515-017-0230-5.

Dietrich, C., Tartler, R., Schröder-Preikshat, W., Lohmann, D., 2012. Understanding linux feature distribution. In: Workshop on Modularity in Systems Software. MODULARITY, ACM, pp. 15–20. http://dx.doi.org/10.1145/2162024.2162030.

Diskin, Z., Maibaum, T., Wassyng, A., Wynn-Williams, S., Lawford, M., 2018. Assurance via model transformations and their hierarchical refinement. In: International Conference on Model Driven Engineering Languages and Systems. MODELS, ACM, pp. 426–436. http://dx.doi.org/10.1145/3239372.3239413.

Dubinsky, Y., Rubin, J., Berger, T., Duszynski, S., Becker, M., Czarnecki, K., 2013. An exploratory study of cloning in industrial software product lines. In: European Conference on Software Maintenance and Reengineering. CSMR, pp. 25–34. http://dx.doi.org/10.1109/CSMR.2013.13.

El-Sharkawy, S., Krafczyk, A., Schmid, K., 2017. An empirical study of configuration mismatches in linux. In: International Systems and Software Product Line Conference. SPLC, ACM, pp. 19–28. http://dx.doi.org/10.1145/3106195.3106208.

European Committee for Electrotechnical Standardization, 2018. Railway applications - communication, signaling and processing systems - safety related electronic systems for signaling, standard. Nov.

Filipovikj, P., Rodriguez-Navas, G., Nyberg, M., Seceleanu, C., 2018. Automated SMT-based consistency checking of industrial critical requirements. SIGAPP Appl. Comput. Rev. 17 (4), 15–28.

Fogdal, T., Scherrebeck, H., Kuusela, J., Becker, M., Zhang, B., 2016. Ten years of product line engineering at danfoss: lessons learned and way ahead. In: International Systems and Software Product Line Conference. SPLC, ACM, pp. 252–261. http://dx.doi.org/10.1145/2934466.2934491.

Gaeta, J.P., Czarnecki, K., 2015. Modeling aerospace systems product lines in SysML. In: International Conference on Software Product Lines. SPLC, ACM, pp. 293–302. http://dx.doi.org/10.1145/2791060.2791104.

Gallina, B., Gallucci, A., Lundqvist, K., Nyberg, M., 2013. VROOM & cC: a method to build safety cases for ISO 26262-compliant product lines. In: Conference on Computer Safety, Reliability and Security (SAFECOMP): SASSUR Workshop. HAL.

Gallina, B., Javed, M.A., Muram, F.U., Punnekkat, S., 2012. A model-driven dependability analysis method for component-based architectures. In: Euromicro Conference on Software Engineering and Advanced Applications. SEAA, IEEE, pp. 233–240. http://dx.doi.org/10.1109/SEAA.2012.35.

Gallina, B., Kashiyarandi, S., Zugsbratl, K., Geven, A., 2014. Enabling cross-domain reuse of tool qualification certification artefacts. In: International Conference on Computer Safety, Reliability and Security. SAFECOMP, Springer, pp. 255–266. http://dx.doi.org/10.1007/978-3-319-10557-4_28.

Gomaa, H., 2011. Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures. Cambridge University Press, http://dx.doi.org/10.1017/CBO9780511779183.

Górski, J., Jarzębowicz, A., Miler, J., et al., 2012. Supporting assurance by evidence-based argument services. In: International Conference on Computer Safety, Reliability, and Security. SAFECOMP, Springer, pp. 417–426.

Graf, S., Quinton, S., 2007. Contracts for BIP: Hierarchical interaction models for compositional verification. In: Forum for Fundamental Research on Theory, Models, and Application for Distributed Systems. Springer, pp. 1–18. http://dx.doi.org/10.1007/978-3-540-73196-2_1.

Gurov, D., Lidström, C., Nyberg, M., Westman, J., 2017. Deductive functional verification of safety-critical embedded C-code: An experience report. In: Critical Systems: Formal Methods and Automated Verification Workshop. Springer, pp. 3–18.

Habli, I., 2009. Model-Based Assurance of Safety-Critical Product Lines (Ph.D. thesis). Department of Computer Science, University of York, York, UK.

Habli, I., Ibarra, I., Rivett, R.S., Kelly, T., 2010. Model-Based Assurance for Justifying Automotive Functional Safety. Tech. Rep., SAE Technical Paper.

Habli, I., Kelly, T., 2006. Process and product certification arguments: Getting the balance right. SIGBED Rev. 3 (4), http://dx.doi.org/10.1145/1183088.1183090.

Habli, I., Kelly, T., 2010. A safety case approach to assuring configurable architectures of safety-critical product lines. In: International Symposium on Architecting Critical Systems. ISARC 10, Springer, pp. 142–160. http://dx.doi.org/10.1007/978-3-642-13556-9_9.

Haugen, Ø., Øgård, O., 2014. BVR – better variability results. In: System Analysis and Modeling: Models and Reusability. Springer, pp. 1–15. http://dx.doi.org/10.1007/978-3-319-11743-0_1.

Hawkins, R., Habli, I., Kolovos, D., Paige, R., Kelly, T., 2015. Weaving an assurance case from design: a model-based approach. In: International Symposium on High Assurance Systems Engineering. HASE, IEEE, pp. 110–117. http://dx.doi.org/10.1109/HASE.2015.25.

Hubaux, A., Xiong, Y., Czarnecki, K., 2012. A user survey of configuration challenges in Linux and eCos. In: International Workshop on Variability Modeling of Software-Intensive Systems. VAMOS, ACM, pp. 149–155. http://dx.doi.org/10.1145/2110147.2110164.

Hutchesson, S., McDermid, J., 2010. Development of high-integrity software product lines using model transformation. In: Computer Safety, Reliability, and Security. SAFECOMP, Springer, pp. 389–401.

Hutchesson, S., McDermid, J., 2011. Towards cost-effective high-assurance software product lines: The need for property-preserving transformations. In: International Software Product Line Conference. SPLC, IEEE, pp. 55–64. http://dx.doi.org/10.1109/SPLC.2011.32.

Hutchesson, S., McDermid, J., 2013. Trusted product lines. Inf. Softw. Technol. 55, 525–540. http://dx.doi.org/10.1016/j.infsof.2012.06.005.

International Organization for Standardization, 2011. ISO 26262: Road vehicles - functional safety. Nov.

Jackson, D., 2002. Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. (TOSEM) 11 (2), 256–290.

Javed, M.A., Gallina, B., 2018. Safety-oriented process line engineering via seamless integration between EPF composer and BVR tool. In: International Systems and Software Product Line Conference, Vol. 2. SPLC, ACM, pp. 23–28. http://dx.doi.org/10.1145/3236405.3236406.

Javed, M.A., Gallina, B., Carlsson, A., 2019. Towards variant management and change impact analysis in safety-oriented process-product lines. In: Symposium on Applied Computing. SAC, ACM, pp. 2372–2375. http://dx.doi.org/10.1145/3297280.3297634.

Junior, E.A.O., Gimenes, I.M.S., Maldonado, J.C., 2010. Systematic management of variability in UML-based software product lines. J. Univers. Comput. Sci. 16 (17), 2374–2393. http://dx.doi.org/10.3217/jucs-016-17-2374.

Kästner, C., Apel, S., 2008. Integrating compositional and annotative approaches for product line engineering. In: Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering. pp. 35–40.

Kelly, T., Bates, S., 2005. The costs, benefits, and risks associated with pattern based and modular safety case development. In: UK MoD Equipment Safety Assurance Symposium. ESAS, UK Ministry of Defense.

Kelly, T.P., McDermid, J.A., 1997. Safety case construction and reuse using patterns. In: Computer Safety, Reliability, and Security. SAFECOMP, Springer, pp. 55–69. http://dx.doi.org/10.1007/978-1-4471-0997-6_5.

Liebig, J., Apel, S., Lengauer, C., Kästner, C., Schulze, M., 2010. An analysis of the variability in forty preprocessor-based software product lines. In: International Conference on Software Engineering-Volume. ICSE, ACM, pp. 105–114. http://dx.doi.org/10.1145/1806799.1806819.

Linden, F., Schmid, K., Rommes, E., 2007. Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering. Springer, http://dx.doi.org/10.1007/978-3-540-71437-8.

Mariani, L., Papagiannakis, S., Pezze, M., 2007. Compatibility and regression testing of COTS-component-based software. In: International Conference on Software Engineering. ICSE, IEEE, pp. 85–95. http://dx.doi.org/10.1109/ICSE.2007.26.

Mendonca, M., Wąsowski, A., Czarnecki, K., 2009. SAT-based analysis of feature models is easy. In: International Software Product Line Conference. SPLC, ACM, pp. 231–240. http://dx.doi.org/10.5555/1753235.1753267.

Metzger, A., Pohl, K., 2014. Software product line engineering and variability management: achievements and challenges. In: Proceedings of the Future of Software Engineering. FOSE, ACM, pp. 70–84. http://dx.doi.org/10.1145/2593882.2593888.

Meyer, B., 1992. Applying "design by contract". Computer 25 (10), 40–51. http://dx.doi.org/10.1109/2.161279.

Mukelabai, M., Nešić, D., Maro, S., Berger, T., Steghöfer, J.-P., 2018. Tackling combinatorial explosion: A study of industrial needs and practices for analyzing highly configurable systems. In: International Conference on Automated Software Engineering. ASE, ACM, pp. 155–166. http://dx.doi.org/10.1145/3238147.3238201.

Muram, F.U., Gallina, B., Rodríguez, L.G., 2018. Preventing omission of key evidence fallacy in process-based argumentations. In: International Conference on the Quality of Information and Communications Technology. QUATIC, IEEE, pp. 65–73. http://dx.doi.org/10.1109/QUATIC.2018.00019.

Nešić, D., Online appendix. http://dx.doi.org/10.5281/zenodo.4560657.

Nešić, D., Krüger, J., Stănciulescu, c., Berger, T., 2019. Principles of feature modeling. In: Foundations of Software Engineering. FSE, ACM, pp. 62–73. http://dx.doi.org/10.1145/3338906.3338974.
Nešić, D., Nyberg, M., 2018. Verifying contract-based specifications of product lines using description logic. In: International Workshop on Description Logic, DL. p. 13.
Nešić, D., Nyberg, M., Barbara, G., 2019. Constructing product-line safety cases using contract-based specifications. In: International Symposium on Applied Computing. SAC, ACM, pp. 2022–2031. http://dx.doi.org/10.1145/3297280.3297479.
Object Management Group, 2008. Software & systems process engineering metamodel. SPEM 2.0.
Object Management Group, 2019. Structured assurance case metamodel, standard. SACM 2.1, Beta version.
de Oliveira, A.L., Braga, R.T.V., Masiero, P.C., Papadopoulos, Y., Habli, I., Kelly, T., 2015. Supporting the automated generation of modular product line safety cases. In: International Conference on Dependability and Complex Systems. DePCoS-RELCOMEX, Springer, pp. 319–330. http://dx.doi.org/10.1007/978-3-319-19216-1_30.
Origin Consulting (York) Limited, 2018. GSN community standard version 2. Jan.
Rhein, A.v., Grebhahn, A., Apel, S., Siegmund, N., Beyer, D., Berger, T., 2015. Presence-condition simplification in highly configurable systems. In: International Conference on Software Engineering. ICSE, ACM, pp. 178–188. http://dx.doi.org/10.1109/ICSE.2015.39.
Rhodes, T., Boland, F., Fong, E., Kass, M., 2010. Software assurance using structured assurance case models. J. Res. Natl. Inst. Stand. Technol. 115 (3), 209. http://dx.doi.org/10.6028/jres.115.013.
Rosen, K.H., Krithivasan, K., 2012. Discrete Mathematics and its Applications: With Combinatorics and Graph Theory. Tata McGraw-Hill Education.
Rosenmüller, M., Siegmund, N., Apel, S., Saake, G., 2011. Flexible feature binding in software product lines. Autom. Softw. Eng. 18, 163–197. http://dx.doi.org/10.1007/s10515-011-0080-5.
Runeson, P., Host, M., Rainer, A., Regnell, B., 2012. Case Study Research in Software Engineering: Guidelines and Examples. John Wiley & Sons.
Sljivo, I., Gallina, B., Carlson, J., Hansson, H., 2016. Configuration-aware contracts. In: International Conference on Computer Safety, Reliability, and Security. SAFECOMP, Springer, pp. 43–54. http://dx.doi.org/10.1007/978-3-319-45480-1_4.
Sljivo, I., Gallina, B., Carlson, J., Hansson, H., Puri, S., 2017. A method to generate reusable safety case argument-fragments from compositional safety analysis. J. Syst. Softw. 131, 570–590. http://dx.doi.org/10.1016/j.jss.2016.07.034.
Svendsen, A., Zhang, X., Lind-Tviberg, et al., 2010. Developing a software product line for train control: A case study of CVL. In: International Conference on Software Product Lines. SPLC, Springer, pp. 106–120. http://dx.doi.org/10.1007/978-3-642-15579-6_8.
The International Electrotechnical Commission, 2010. ISO 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems.
Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G., 2014. A classification and survey of analysis strategies for software product lines. ACM Comput. Surv. 47 (1), 6. http://dx.doi.org/10.1145/2580950.
UL4600 Task Group, 2020. UL4600 - Standard for the Evaluation of Autonomous Products. Tech. Rep., Underwriters Laboratories (UL).
de la Vara, J.L., Ruiz, A., Gallina, B., et al., 2019. The AMASS approach for assurance and certification of critical systems. In: Embedded World 2019.
Wei, R., Kelly, T.P., Dai, X., Zhao, S., Hawkins, R., 2019. Model based system assurance using the structured assurance case meta- model. J. Syst. Softw. 154, 211–233. http://dx.doi.org/10.1016/j.jss.2019.05.013.
Westman, J., Nyberg, M., 2015. Contracts for Specifying and Structuring Requirements on Cyber-Physical Systems. CRC Press (Ch. 13).
Westman, J., Nyberg, M., 2018a. Conditions of contracts for separating responsibilities in heterogeneous systems. Form. Methods Syst. Des. 52 (2), 147–192. http://dx.doi.org/10.1007/s10703-017-0294-7.
Westman, J., Nyberg, M., 2018b. Preserving contract satisfiability under non-monotonic composition. In: Formal Techniques for Distributed Objects, Components, and Systems. FORTE, Springer, pp. 181–195. http://dx.doi.org/10.1007/978-3-319-92612-4_10.
Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J., 2009. Formal methods: Practice and experience. ACM Comput. Surv. 41 (4), http://dx.doi.org/10.1145/1592434.1592436.
Wozniak, L., Clements, P., 2015. How automotive engineering is taking product line engineering to the extreme. In: International Conference on Software Product Line. SPLC, ACM, pp. 327–336. http://dx.doi.org/10.1145/2791060.2791071.
Ziadi, T., Jezequel, J.-M., 2006. Software product line engineering with the UML: Deriving products. In: Software Product Lines. Springer, pp. 557–588. http://dx.doi.org/10.1007/978-3-540-33253-4_15.

Damir Nešić is a Ph.D. candidate at the division of Mechatronics at the Royal Institute of Technology in Stockholm. Before starting his Ph.D. project, he has spent one semester as a researcher at the Mälardalen University. Prior to this, he has spent several years as an embedded-systems development engineer in the medical industry. He has received his Master in Microsystems and Microelectronics from the University of Niš, Serbia.

Mattias Nyberg is an Adjunct Professor in Dependable Control Systems at the division of Mechatronics at the Royal Institute of Technology in Stockholm. Simultaneously, he is an Expert Engineer at Division of Systems Architecture in the global manufacturer of heavy vehicles, Scania. Previously he has been employed as an Associate Professor at Linköping University, and also at the DaimlerChrysler Research Division in Stuttgart. He received his Master in Computer Science from Linköping University, and a Master in Electrical Engineering and Applied Physics from the Case Western Reserve University. He got his Ph.D. in Electrical Engineering from Linköping University.

Barbara Gallina is an Associate Professor of Dependable Software Engineering at Mälardalen University. She is Vice-chair of the security subgroup within EWICS and member of IEEE SMC Technical Committee on Homeland Security. Within AMASS, a large EU-ECSEL funded project, she played various roles: project technical-manager, work package leader, task leader, and land coordinator. She also led dependability-related work packages in the EU-Artemis funded SafeCer and CONCERTO projects. She has received a M.Sc. in Computer Engineering and a II-level Master in IT, both from Politecnico di Milano (Italy). She got her Ph.D. in Computer Science from the University of Luxembourg.