Requirements Engineering
https://doi.org/10.1007/s00766-017-0286-6

ORIGINAL ARTICLE

Providing tool support for specifying safety-critical systems by enforcing syntactic contract conditions

Jonas Westman[1] · **Mattias Nyberg**[1]

Received: 22 September 2016 / Accepted: 18 December 2017

Abstract
Functional safety standards such as IEC 61508 and ISO 26262 advocate a particularly stringent *requirements engineering* where safety requirements must be *structured* in a hierarchical manner and *specified* in accordance with the system *architecture*. In contrast to the stringent requirements engineering in functional safety standards, according to previous studies, requirements engineering in industry is in general of poor quality. *Contracts theory* has been previously shown to be suitable for supporting such a stringent requirements engineering effort; this support has also been implemented in tools. However, to use these contract-based tools, requirements must be formalized, which is a major challenge in industry. Therefore, to support current industrial requirements engineering practice and the stringent requirements engineering in functional safety standards, it is shown how tool support can be provided even when requirements, and also architectures, are not formalized. This is achieved by enforcing *syntactic*, yet formal, conditions in contracts theory. Despite the need for further validation, initial findings in an industrial case study indicate high potential in realizing the proposed support in an industrial setting.

Keywords Syntactic · Contracts · Conditions · Requirements engineering · Specification · Structuring · Authoring · Safety · FuSa · Architecture · IEC 61508 · ISO 26262 · Elements · Compositional · Tool Support

1 Introduction

Requirements engineering (*RE*) [12, 35] is a well-established and recommended practice within the field of systems engineering. RE is particularly emphasized for achieving *functional safety* (FuSa), i.e., absence of unreasonable risk due to failures of electrical/electronic (E/E)-systems [37]. In fact, the general FuSa standard IEC 61508 [36] advocates that requirements should form the backbone of a structured argumentation for the FuSa of an overall system. In such a structured argumentation, each requirement is a *safety requirement*, i.e., a requirement with a safety integrity level (SIL) [22, 36, 37, 54] that specifies the required *reliability* [63] of a system or component, in order to achieve a tolerable level of risk.

✉ Jonas Westman
jowestm@kth.se
https://www.kth.se/profile/jowestm

Mattias Nyberg
matny@kth.se

[1] Machine Design, Kungliga Tekniska Högskolan (KTH), Brinellvägen 83, 100 44 Stockholm, Sweden

In IEC 61508 and its derivative FuSa standards such as ISO 26262 [37] for the automotive domain, safety requirements must be structured in a hierarchical manner in accordance with the *system architecture* [38]: at each level, safety requirements must be *allocated* to *architecture elements* and *trace links* [15] must be established between requirements on different levels. An intended property characterized by this manner of structuring requirements is *completeness*, i.e., 'the safety requirements at one level fully implement all safety requirements of the previous level' [37]. This is a property that also must be verified, and thus, a high degree of *stringency* is required when specifying requirements, their allocation, and their hierarchical structure.

Despite the demand on highly stringent RE to achieve FuSa, requirements in industry are in general of poor quality [2] and are typically incomplete [25], and this is also true for safety requirements [25, 49]. Considering a typical RE tool such as IBM Rational DOORS, other than basic impact analyzes, the tool neither gives feedback nor guides a user when specifying, allocating, and structuring requirements; thus, a property such as completeness must be established without any concrete support from the tool. The view in [70], which is shared in the present paper, is that RE is a

Published online: 30 December 2017

complex and error-prone process that can benefit from more intelligent tool support in general. In fact, in order to comply with FuSa standards that require a particularly stringent RE, tool support, which gives feedback to and guides the user when specifying a system, is crucial.

Therefore, the present paper presents novel ideas on how such tool support can be provided when authoring specifications for a system in an industrial setting. This support is provided by applying the work in [6, 68, 73, 75, 76] that present a formal and general *contracts* [53] theory for modeling and specifying systems.

In particular, this contracts theory contains a concept called a *contract structure* that captures a hierarchical structuring of requirements based on a formal interpretation of completeness. Thus, establishing a contract structure sets a basis for achieving the stringent RE effort advocated by FuSa standards. Establishing a contract structure with the intent of achieving completeness consists of the tasks of specifying:

(I) allocation of requirements to architecture elements;
(II) architecture element interfaces consisting of *port variables*;
(III) requirements; and
(IV) trace links between requirements.

These tasks (I)–(IV) are also described in FuSa standards; however, the fact is that in contracts theory [6, 68, 73, 75, 76], these tasks are given *formal semantics*, i.e., interpretations in formal (mathematical) conditions. The present paper capitalizes on this fact by considering the support that can be provided for tasks (I)–(IV) by having a tool that evaluates these *formal contract conditions*.

Notably, there already exist approaches such as [13, 14, 16] where formal contract conditions are evaluated in tools. However, the tool support in these approaches is dependent on that contracts must be formally represented in the language linear-time temporal logic (LTL) [59]. Despite the fact that formal representations have several advantages over non-formal ones, formal languages are difficult to use by non-experts [23] and in industrial practice, 'overcoming the burden of formalization is a major challenge' [10]. Therefore, instead of focusing on enforcing all of the contract conditions in [6, 68, 73, 75, 76], the present paper instead identifies necessary conditions of the formal interpretations of tasks (I)–(IV) where these conditions can be evaluated even when requirements and architectures are not represented formally; in the following, such necessary conditions will be called *syntactic contract conditions*.

The rest of this paper is organized as follows. Section 1.1 provides an overview of the contributions and validation approach of the paper. Section 2 introduces relevant parts of the contracts theory in [6, 68, 73, 75, 76]. Section 3 identifies syntactic contract conditions of tasks (I)–(IV) in this contracts theory, and as the main contribution, shows how support can be provided for tasks (I)–(IV) by enforcing these syntactic contract conditions. Section 4 then describes and draws conclusions from the industrial case study. Section 5 discusses related work, and Sect. 6 summarizes the present paper and draws conclusions.

1.1 Contributions and validation

As the main contribution, the present paper provides support for tasks (I)–(IV) by evaluating their syntactic contract conditions. The proposed support is presented in the context of authoring specifications, containing requirements and interface descriptions, and that are structured in accordance with the architecture of a system. In the considered context, the proposed support is shown to provide feedback and guidance to a user authoring a specification. This sort of support for tasks (I), (II), and (IV) is shown to be possible to provide regardless of how requirements are represented in specifications and when architectures are represented formally or *semiformally*, as an hierarchy of interfaces consisting of port variables. For task (III), feedback and guidance can be provided when architectures are represented formally or semiformally and when requirements are represented formally or *semiformally*, i.e., as free text with distinguishable port variables.

Considering the generality of the underlying contract theory, the proposed support is applicable to systems in any domain, e.g., software (SW), hardware (HW), mechanical, electrical, etc, and also to *heterogeneous systems* [31, 48, 64], i.e., systems composed of parts from multiple domains. The applicability of the proposed support is indeed also not limited to any type of design flow; that is, the support prescribes neither a certain order in which specifications are authored nor an order in which content is entered within a specification. This flexibility in design flow caters to the fact that different developers have dissimilar, yet successful, approaches for optimizing systems development. For example, in some cases, e.g., when development is outsourced, a top-down design flow might be optimal; in other cases, a bottom-up flow or a hybrid approach is more suitable.

As an initial form of validation, an industrial case study was performed where the proposed support was implemented and integrated into the development tool chain of Scania—a global heavy trucks manufacturer located in Sweden. Despite the need for further evaluations, especially long-term empirical studies, the case study did not only indicate a strong potential in realizing the proposed support in an industrial setting, but also that solutions, needed for realizing the support, could by themselves increase the quality of specifications. For example, a key concept used for realizing this support was Linked Data [9], which was used to

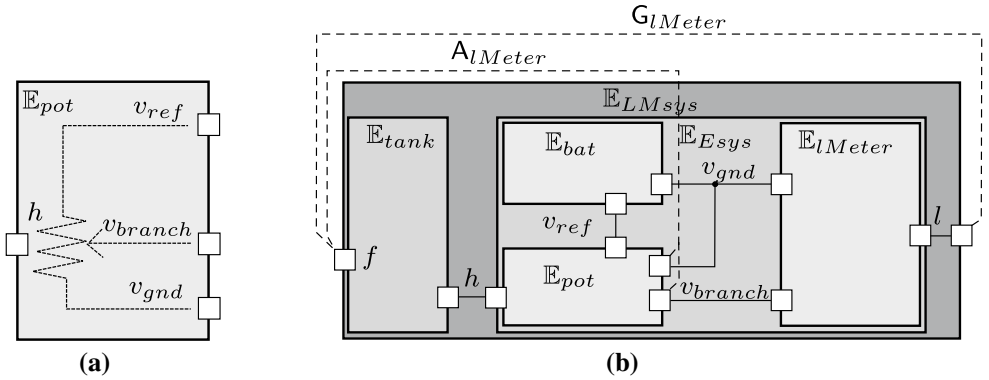

Fig. 1 In **a**, an element $\mathbb{E}_{pot} = (X_{pot}, \mathsf{B}_{pot})$, representing a potentiometer, is shown. In **b**, an architecture, representing a 'Level Meter system' and its parts, and a contract $C_{lMeter} = (\{\mathsf{A}_{lMeter}\}, \mathsf{G}_{lMeter}, X_{lMeter})$, are shown

enable *formal referencing* in between specifications and to architecture data. In addition to providing a foundation for formal referencing, as shown in the present paper, Linked Data are also able to ensure consistency of data presented in specifications and can, therefore, be used to increase quality of specification in general.

2 Contracts theory

The notion of *contracts* was first introduced in [53] for formal specification of SW. However, the principles behind contracts can be traced back to early ideas on *compositional* [33, 66] proof-methods [32, 41, 55]. In [6, 68, 76], the use of contracts is extended from formal specification of SW to serving as a central systems engineering philosophy to support the design of systems in general. The work in [73] incorporates the work in [6, 68, 76] and presents a contracts theory that introduces new concepts such as *architecture*. This contracts theory is extended in [75] to a safety-critical context with the notion of SILs.

Section 3 will later show how support can be provided for tasks (I)–(IV) by enforcing conditions from the contracts theory in [6, 68, 73, 75, 76]. The present section summarizes established conditions and related notions from this contracts theory. To match the context in the present paper, these conditions and notions are sometimes presented from slightly different perspectives than in [6, 68, 73, 75, 76]. Due to this difference in perspectives, and to make the present paper self-contained, this section is quite thorough in introducing these notions and conditions.

2.1 Assertions, elements, and architectures

The theory [6, 68, 73, 75, 76] relies on a general formalism called *assertions* for characterizing requirements and behaviors. Formally, an assertion is a set of *runs*, i.e., value sequences, over a universal set of variables Ξ.

Despite their runs being over Ξ, assertions can be syntactically represented by *constraints*, e.g., by equations, inequalities, or logical formulas, expressed over a *subset* of Ξ. For example, an assertion W', represented by equation $u = v$, is the set of all runs that are over Ξ and that are solutions to $u = v$. The necessary and sufficient set of variables to syntactically represent an assertion W is called *the set of variables constrained by* W and is denoted X_{W}, e.g., the set $X_{\mathsf{W}'} = \{u, v\}$ is the set of variables constrained by W'.

The concept of *elements* [73, 75, 76] generalizes Heterogeneous Rich Components (HRCs) [8, 18, 42] as described in [6, 68] and is used to represent any functional, logical, or technical design entity of a heterogeneous system, e.g., as a Systems Modeling Language (SysML) block [26]. Formally, an *element* $\mathbb{E}$ is an ordered pair (X, B) where:

- X is a set of variables called the *interface of* $\mathbb{E}$ where each $x \in X$ is called *a port variable*; and
- B is an assertion, called the *behavior* of $\mathbb{E}$, such that the set of variables constrained by B is a subset of X.

As an illustrative example of an element, let $\mathbb{E}_{pot} = (X_{pot}, \mathsf{B}_{pot})$ be an element representing a potentiometer. The element and its port variables are shown in Fig. 1a as a rectangle filled with gray and white boxes on the edges of the rectangle, respectively. The port variables v_{ref}, v_{branch}, and v_{gnd} represent the reference, branch, and ground voltages, respectively. Furthermore, h represents the position (0–100%) of the 'slider' that moves over the resistor and branches the circuit. Given a representation where it is assumed that the branched circuit is connected to a resistance that is significantly larger than the resistance of the potentiometer, the behavior B_{pot} can be syntactically represented by equation $h = \frac{v_{branch} - v_{gnd}}{v_{ref} - v_{gnd}}$.

A set of elements can be organized into an *architecture* [73, 75, 77], which describes a hierarchical nesting of elements. This hierarchical nesting can be viewed as a rooted tree; thus, in the following, terminology from graph theory [21] will be used to describe positions of elements in an architecture, relative to each other. The underlying principle is to combine individual behaviors using *intersection* where the *sharing of port variables* between elements captures the interaction points between the elements.

Prior to presenting a formal definition of architecture, the concept is introduced by considering a set of elements representing the parts of a 'Level Meter system' (LM-system) as shown in Fig. 1b. The sharing of a port variable between elements is shown as either by a line connecting two or more boxes corresponding to the same port variable or by the appearance of the same box on edges of several rectangles.

The LM-system $\mathbb{E}_{LMsys}$ consists of a tank $\mathbb{E}_{tank}$ and an electric-system $\mathbb{E}_{Esys}$, which further consists of the potentiometer $\mathbb{E}_{pot}$ shown in Fig. 1a, a battery $\mathbb{E}_{bat}$, and a level meter $\mathbb{E}_{lMeter}$. The slider h of $\mathbb{E}_{pot}$ is connected to a 'floater,' trailing the level f in the tank. In this way, the potentiometer $\mathbb{E}_{pot}$ is used as a level sensor to estimate the level in the tank. The estimated level is presented by the level meter $\mathbb{E}_{lMeter}$ where l denotes the presented level.

The behaviors B_{bat}, B_{lMeter}, and B_{tank} of the leaf elements $\mathbb{E}_{bat}$, $\mathbb{E}_{lMeter}$, and $\mathbb{E}_{tank}$ are represented by equations $v_{ref} - v_{gnd} = 5V$, $l = \frac{v_{branch} - v_{gnd}}{5V}$, and $h = f$, respectively. The behavior of a non-leaf element (X, B) is the assertion that is the *projection* [73, 75, 78] of the intersection of the behaviors of its children $\{(X_i, \mathsf{B}_i)\}_{i=1}^{N}$ onto the interface X; syntactically, B is represented by the constraints, representing $\bigcap_{i=1}^{N} \mathsf{B}_i$, limited to be only over X. Formally, this is represented as $\mathsf{B} = \widehat{proj}_X(\bigcap_{i=1}^{N} \mathsf{B}_i)$ where $\widehat{proj}_X$ denotes *extended projection* [73, 74]. For example, the behavior of $\mathbb{E}_{Esys}$ is $\widehat{proj}_{\{l,h\}}(\mathsf{B}_{bat} \cap \mathsf{B}_{Lmeter} \cap \mathsf{B}_{pot})$, which can be syntactically represented by equation $l = h$.

Definition 1 (*Architecture*) An *architecture*, denoted $\mathscr{A}$, is a set of elements organized into a rooted tree, such that:

(a) for any non-leaf node $\mathbb{E} = (X, \mathsf{B})$, with children $\{(X_i, \mathsf{B}_i)\}_{i=1}^{N}$, it holds that

(i) $X \subseteq \bigcup_{i=1}^{N} X_i$, and
(ii) $\mathsf{B} = proj_X(\bigcap_{i=1}^{N} \mathsf{B}_i)$; and

(b) if there is a child $\mathbb{E}' = (X', \mathsf{B}')$ and a non-descendant $\mathbb{E}'' = (X'', \mathsf{B}'')$ of $\mathbb{E} = (X, \mathsf{B})$, such that $x \in X'$ and $x \in X''$, then it holds that $x \in X$.

The environment of an element $\mathbb{E}$ *in an architecture* $\mathscr{A}$ is denoted $Env_{\mathscr{A}}(\mathbb{E})$ and is the set of elements $\{\mathbb{E}_j\}_{j=1}^{M}$ such that each $\mathbb{E}_j = (X_j, \mathsf{B}_j)$ is either a sibling or a sibling of a proper ancestor of $\mathbb{E}$. Let $\mathsf{B}_{Env_{\mathscr{A}}(\mathbb{E})}$ denote $\bigcap_{j=1}^{M} \mathsf{B}_j$, called the behavior of the environment $Env_{\mathscr{A}}(\mathbb{E})$. □

As an example of an environment of an element in an architecture, the set $\{\mathbb{E}_{tank}\}$ is the environment of $\mathbb{E}_{Esys}$ in the architecture shown in Fig. 1b.

2.2 Contracts

A *contract* [6, 68, 73, 75, 76] $(\{\mathsf{A}_i\}_{i=1}^{N}, \mathsf{G}, X)$ specifies the behavior of an element with an interface X to be such that the guarantee G is fulfilled, given that the assumptions in $\{\mathsf{A}_i\}_{i=1}^{N}$ are fulfilled.

Definition 2 (*Contract*) A *contract* $\mathcal{C}$ is a tuple $(\mathscr{A}, \mathsf{G}, X)$, where

- G is an assertion, called *guarantee*;
- $\mathscr{A}$ is a set of assertions $\{\mathsf{A}_i\}_{i=1}^{N}$ where each A_i is called an *assumption*; and
- X is a set of variables. □

For the sake of readability, let $\mathsf{A}_{\mathscr{A}} = \bigcap_{j=1}^{N} \mathsf{A}_i$.

An element $\mathbb{E} = (X', \mathsf{B})$ is said to *satisfy* [6, 8, 68, 73, 75] a contract $(\mathscr{A}, \mathsf{G}, X)$ if

$$\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G}, \text{ and} \tag{1}$$

$$X = X' . \tag{2}$$

Referring to *a contract for an element*, characterizes the *intent* that the element is to satisfy the contract.

As an illustrative example of a contract, let $(\{\mathsf{A}_{lMeter}\}, \mathsf{G}_{lMeter}, X_{lMeter})$ be a contract $\mathcal{C}_{lMeter}$ for the element $\mathbb{E}_{lMeter}$ where the set of port variables constrained by A_{lMeter} and G_{lMeter} are shown as dashed lines in Fig. 1b. The guarantee G_{lMeter}, represented by equation $l = f$, expresses the intent that the indicated level, displayed by the meter, corresponds to the level in the tank. The assumption A_{lMeter} is represented by equation $f = \frac{v_{branch} - v_{gnd}}{5V}$. In accordance with conditions (1) and (2), contract $\mathcal{C}_{lMeter}$ is satisfied by the element $\mathbb{E}_{lMeter}$.

2.3 Hierarchical structuring of requirements using contracts

Consider a scenario where it is infeasible to verify that a contract $\mathcal{C}$ is satisfied by an element $\mathbb{E}$ in an architecture $\mathscr{A}$, due to the complexity of $\mathbb{E}$. A solution to such an issue is to establish contracts for proper descendants of $\mathbb{E}$ until it is possible to verify that a descendant $\mathbb{E}_i$ of $\mathbb{E}$ satisfies $\mathcal{C}_i$ with the intent that:

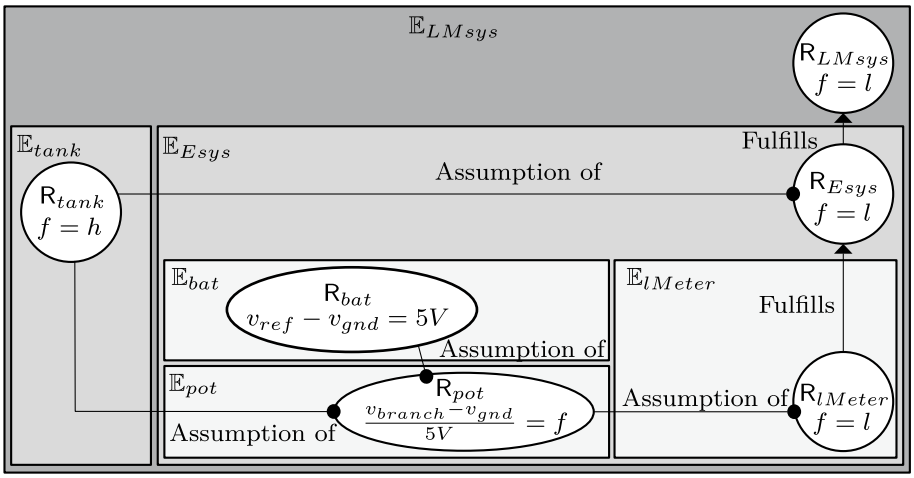

Fig. 2 A contract structure for the architecture shown in Fig. 1b

if each $\mathbb{E}_i$ satisfies C_i then $\mathbb{E}$ satisfies C. (3)

If an architecture only consists of two hierarchical levels, then property (3) corresponds to *dominance/refinement* of contracts as described in [4, 6, 7, 61, 68], the basic idea of *compositionality* [33, 66], and in particular, the notion of *completeness* in ISO 26262 [37]. Thus, property (3) is a central concept in this paper since completeness characterizes the particularly stringent RE advocated in FuSa standards such as IEC 61508 and ISO 26262. Note that there are many reasons for establishing property (3) other than the one considered in the scenario, e.g., to enable parallel development of elements as described in [73].

This section presents concepts originating from [75] that introduces a graph called a *contract structure* that organizes contracts with the intent of achieving property (3). Prior to presenting a formal definition of contract structure in Sect. 2.3.2, an underlying concept of using a contract to express a relation between requirements is described in Sect. 2.3.1. Sufficient conditions on a contract structure to achieve property (3) are then presented in Sect. 2.3.3.

2.3.1 Contracts as requirement relations

Consider contract C_{lMeter} for the element $\mathbb{E}_{lMeter}$, as shown in Fig. 1b. In accordance with Sect. 2.2, the intent is that the behavior of $\mathbb{E}_{lMeter}$ is to be such that the guarantee G_{lMeter} is fulfilled given that the assumption A_{lMeter} is fulfilled. Formulated differently, the guarantee G_{lMeter} is a *requirement* that is *allocated* to $\mathbb{E}_{lMeter}$ with the intent that G_{lMeter} is fulfilled if the assumption A_{lMeter} is fulfilled. This view is in accordance with [78] where guarantees are used to express safety requirements on elements.

Suppose that there is a scenario where the environment, which the element $\mathbb{E}_{Lmeter}$ is to be deployed in, is unknown, e.g., when developing $\mathbb{E}_{Lmeter}$ 'out-of-context' [37]. The assumption A_{Lmeter} of the contract shown in Fig. 1b hence expresses the conditions that the environment of $\mathbb{E}_{Lmeter}$ is to fulfill in *an arbitrary architecture containing* $\mathbb{E}_{Lmeter}$. However, in a *specific architecture*, such as the architecture shown in Fig. 1b, *the assumption* A_{Lmeter} *can rather be seen as a requirement* that is allocated to the potentiometer $\mathbb{E}_{pot}$. This was also observed in [72, 78] where, in the context of an architecture, assumptions are in fact references to other guarantees.

Therefore, in the definition of a contract structure for a *specific architecture* in Sect. 2.3.2, an assumption of a contract for an element $\mathbb{E}$ will correspond to a guarantee of a contract for an element in the environment of $\mathbb{E}$. Formulated differently, *the assumption of a contract for* $\mathbb{E}$ *is a requirement allocated to an element in the environment of* $\mathbb{E}$, *while the guarantee is a requirement allocated to* $\mathbb{E}$. To capture the cases where the use of explicit assumptions are indeed needed, e.g., when it is necessary to express that an assumption is to be fulfilled by two or more guarantees, contract structures can be trivially extended as described in [73].

2.3.2 Contract structure for architecture

Consider a set of contracts $\{C_{LMsys}, C_{Esys}, C_{pot}, C_{bat}, C_{lMeter}\}$ where the guarantee of each contract C_i is a requirement R_i allocated to an element $\mathbb{E}_i$ in the architecture shown in Fig. 1b. Now consider Fig. 2 where these guarantees are structured as nodes in an edge-labeled directed graph as an overlay onto the hierarchical structure of the elements in this architecture.

A guarantee R_i in a contract C_i is an assumption of another contract C_j, if there exists an arc labeled 'Assumption of' from R_i to R_j, visualized as a line with a circle filled with black at the end. For example, the arc from R_{pot} to R_{lMeter} represents that R_{pot} is an assumption of contract C_{lMeter}.

As also shown in Fig. 2, an incoming arc labeled 'Fulfills,' visualized as an arrow, to a guarantee R_i of a contract

for an element $\mathbb{E}_i$ from a guarantee R_j of a contract for a child $\mathbb{E}_j$ of $\mathbb{E}_i$, represents the intent of $\mathsf{R}_j \subseteq \mathsf{R}_i$. For example, the arc from the guarantee R_{Esys} to R_{LMsys} represents the intent of $\mathsf{R}_{Esys} \subseteq \mathsf{R}_{LMsys}$.

Now that the concept of a contract structure for architecture has been introduced informally, the formal definition follows. The following definition is a simplification of a definition in [74], which is in turn based on a definition in [75] and concepts in [73].

Definition 3 (*Contract structure for architecture*) Given an architecture $\mathscr{A}$ and a set $\bigcup_{i=1}^{N}\{(\mathcal{A}_{i,1}, \mathsf{R}_{i,1}, X_i), \ldots, (\mathcal{A}_{i,N_i}, \mathsf{R}_{i,N_i}, X_i)\}$ where each ordered set $(\mathcal{A}_{i,j}, \mathsf{R}_{i,j}, X_i)$ is a contract for an element $\mathbb{E}_i$ of $\mathscr{A}$ and where each assumption in each set $\mathcal{A}_{i,j}$ is either:

(a) a guarantee of a contract for a sibling of $\mathbb{E}_i$; or
(b) an assumption of a contract for a proper ancestor of $\mathbb{E}_i$,

then a *contract structure* for $\mathscr{A}$ is an arc-labeled directed acyclic graph (DAG), such that:

(i) the guarantees $\mathsf{R}_{i,j}$ are the nodes in the DAG;
(ii) each arc is uniquely labeled either 'Assumption of' or 'Fulfills';
(iii) there is an arc labeled 'Assumption of' from a node $\mathsf{R}_{k,l}$ to $\mathsf{R}_{i,j}$, if and only if $\mathsf{R}_{k,l}$ is in $\mathcal{A}_{i,j}$;
(iv) if there is an arc labeled 'Fulfills' from $\mathsf{R}_{i,j}$ to $\mathsf{R}_{k,l}$, then $\mathsf{R}_{k,l}$ is a guarantee of a contract for a proper ancestor of $\mathbb{E}_i$; and
(v) if a guarantee $\mathsf{R}_{i,j}$ is reachable from an assumption A of a contract for a proper ancestor $\mathbb{E}_m$ of $\mathbb{E}_i$, then A is also an assumption of any contract $(\mathcal{A}_{k,l}, \mathsf{R}_{k,l}, X_k)$ where $\mathbb{E}_k$ is a proper ancestor of $\mathbb{E}_i$ and a descendant of $\mathbb{E}_m$ (including itself) and where $\mathsf{R}_{k,l}$ is reachable from $\mathsf{R}_{i,j}$. □

As discussed in Sect. 2.3.1 and as also shown in Fig. 2, conditions (a) and (b) of Definition 3 express that an assumption of a contract for an element $\mathbb{E}$ correspond to a guarantee of a contract for an element in the environment of $\mathbb{E}$, i.e., an assumption is either a guarantee of a contract for a sibling of $\mathbb{E}$, or an assumption of a contract for a proper ancestor of $\mathbb{E}$.

Furthermore, as expressed in conditions (i)–(v) of Definition 3 and shown in Fig. 2, each node in a contract structure corresponds to a requirement allocated to an element in $\mathscr{A}$ and each arc either expresses that a requirement is an assumption of a contract or that the intent is that a requirement is to fulfill another requirement.

Consider the contract structure shown in Fig. 3 that is intended to clarify why the graph would not be a contract structure if the dashed arc is added to the graph, as expressed in condition (v) of Definition 3. In Fig. 3, the intent is that the guarantee R_2 is to be fulfilled by the behavior of $\mathbb{E}$, *regardless of its environment*. However, if the dashed arc is added, then the above-mentioned statement is contradicted since the graph then specifies that R_2 is to be fulfilled by the behavior of $\mathbb{E}$, *given that its environment fulfills* R'. This is due to the fact that the graph also specifies that $\mathsf{R}_{1,1}$ is to, together with $\mathsf{R}_{1,2}$, fulfill R_2, and the behavior of the child $\mathbb{E}_1$ of $\mathbb{E}$ is to fulfill $\mathsf{R}_{1,1}$, given that R' is fulfilled.

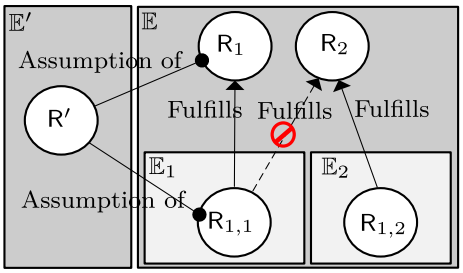

Fig. 3 A contract structure that would not be a contract structure if the dashed arc is added to the graph

For a more detailed explanation of the concept of contract structures, see [74] and also [75] where this concept is also applied in a major industrial case study. Contract structures recast the notion of decomposition structures as presented in [73] in the context of RE concepts and extends the notion from two levels to an arbitrary number. A similar concept to a contract structure is presented in [56, 57] based on Bayesian networks, but with the specific focus to model failure propagation.

2.3.3 Sufficient conditions on requirements in contract structure

This section presents a theorem based on a contract structure where the theorem expresses sufficient conditions for property (3) to hold. This theorem corresponds to a theorem in [74] and formalizes the stringent RE effort in establishing completeness as advocated in FuSa standards such as IEC 61508 and ISO 26262.

Theorem 1 *Given an architecture $\mathscr{A}$ and set of contracts $\mathfrak{C}$ organized as a contract structure $\mathfrak{C}$ for $\mathscr{A}$, it holds that an element $\mathbb{E} \in \mathscr{A}$ satisfies a contract $(\mathcal{A}, \mathsf{R}, X) \in \mathfrak{C}$ for $\mathbb{E} = (X', \mathsf{B})$ if:*

(i) *for each contract $C'' = (\mathcal{A}'', \mathsf{R}'', X'') \in \mathfrak{C}$ for a descendent element $\mathbb{E}''$ of $\mathbb{E}$ where R is reachable from R'' and where R'' does not have any incoming 'Fulfills' arcs, it holds that $\mathbb{E}''$ satisfies C'';*
(ii) it holds that

$$X = X', \text{ and} \tag{4}$$

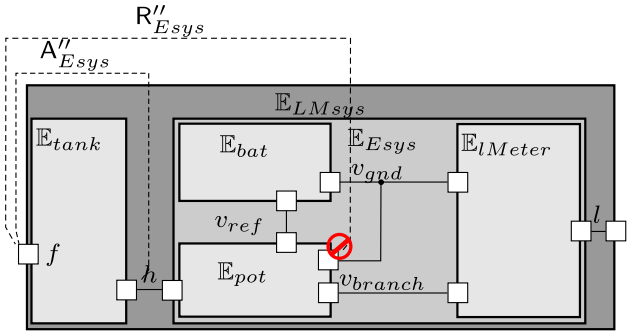

Fig. 4 A contract $C''_{Esys} = (\{\mathsf{A}''_{Esys}\}, \mathsf{R}''_{Esys}, X_{Esys})$ where condition (5) is violated

$$X_{\mathsf{R}} \subseteq X_{Env_{of}(\mathbb{E})} \cup X' , \tag{5}$$

where X' is the interface of $\mathbb{E}$ and where $X_{Env_{of}(\mathbb{E})}$ is the union of the interfaces of the elements in the environment of $\mathbb{E}$; and

(iii) for each contract $C'' = (\mathcal{A}'', \mathsf{R}'', X'') \in \mathfrak{C}$ *for a descendent element $\mathbb{E}''$ of $\mathbb{E}$ where R is reachable from R'', it holds that $\bigcap_{i=1}^{N} \mathsf{R}_i \subseteq \mathsf{R}''$ where $\{\mathsf{R}_1, \ldots, \mathsf{R}_N\}$ is the set of direct predecessors of R'' with 'Fulfills' arcs to R''.*

Condition (i) ensures that the antecedent, i.e., the if-part, of property (3) holds for each contract containing a lowest-level requirement from where R can be reached. Conditions (ii) and (iii) of Theorem 1 are sufficient to ensure property (3). Condition (iii) ensures that all of the 'Fulfills' arcs in paths from lower-level requirements to R, do in fact hold.

Condition (ii) embeds condition (4), which corresponds to condition (2) presented in Sect. 2.2, and also condition (5). Figure 4 shows an example of when condition (5) is violated with respect to contract $C''_{Esys} = (\{\mathsf{A}''_{Esys}\}, \mathsf{R}''_{Esys}, X_{Esys})$ for the element $\mathbb{E}_{Esys}$ in the architecture previously described in Sect. 2.1. In this example, condition (5) is violated due to the fact that the guarantee R''_{Esys} constrains a port variable that cannot be constrained by the behavior of $\mathbb{E}_{Esys}$ nor of its environment in the considered architecture, i.e., it holds $X_{\mathsf{R}''_{Esys}} \not\subseteq X_{tank} \cup X_{Esys}$.

Now, to illustrate the use of Theorem 1, consider the contract structure shown in Fig. 2. Since $X_{\mathsf{R}_{LMsys}} \subseteq X_{LMsys}$ and since the relations $\mathsf{R}_{lMeter} \subseteq \mathsf{R}_{Esys}$ and $\mathsf{R}_{Esys} \subseteq \mathsf{R}_{LMsys}$ hold, it can be inferred, through the use of Theorem 1, that if the leaf elements of the architecture in Fig. 1b satisfy their contracts, then $\mathbb{E}_{LMsys}$ satisfies C_{LMsys}.

Remark 1 (Circular reasoning) Since the assumptions and guarantees of a contract structure are organized into a directed *acyclic* graph, the use of circular argumentation is avoided. Note that circularity can be resolved in other ways, e.g., by introducing assumptions about the computational model [1] or the timing model [52].

2.4 Extending contracts theory with SILs

As described in Sect. 2.3, a contract structure supports a hierarchical structuring of requirements and the individual tracing of lower-level safety requirements to top-level safety requirements. This individual tracing of requirements is needed to comply with, e.g., ISO 26262 where the assignment of SILs to lower-level requirements on components is determined based on their individual tracing to higher-level requirements.

More specifically, in ISO 26262, SILs are assigned to top-level safety requirements. Considering that a SIL was in Sect. 1 described as a measure of the required reliability of a system or component in order to achieve a tolerable level of risk, a SIL for a requirement expresses the tolerable level of risk of violating the requirement. As the safety requirements on a system are broken down into safety requirements on sub-systems, SILs are either inherited from a requirement at a higher level to a requirement at a lower level, or decomposed, where the SIL is lowered, as a result of introducing redundancy into the system. If the sub-systems are sufficiently reliable in fulfilling their respective safety requirements, as specified by the SILs, then it follows that the system is sufficiently reliable in fulfilling the top-level safety requirement.

Exploiting the fact that a contract structure formalizes such a hierarchical structuring of requirements, the work in [75] presents formal definitions for assigning SILs to requirements in a contract structure in accordance with FuSa standards. These definitions of SIL assignment are presented in Appendix. Section 3 will describe shortly how these definitions can be used to also support the assignment of SILs to requirements organized as a contract structure.

3 Support for authoring specifications by enforcing syntactic contract conditions

Section 2 presented a general contracts theory for specifying systems. Considering RE in particular, this contracts theory includes Theorem 1 that can, in combination with Definition 3, i.e., the definition of a contract structure for an architecture, be used for achieving completeness between requirements on different levels in a hierarchy in accordance with property (3). Thus, this definition and theorem formalize the particularly stringent RE advocated in FuSa standards such as IEC 61508 and ISO 26262.

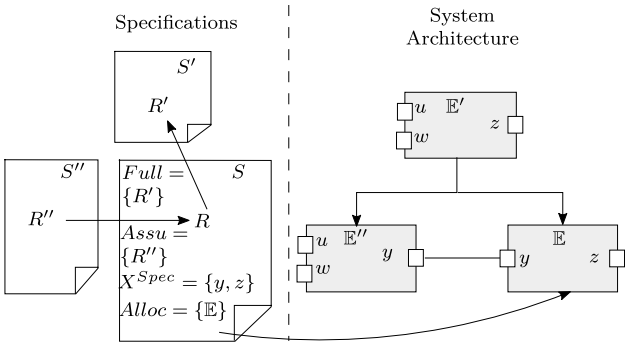

Fig. 5 Three specifications S, S', and S'', structured with respect to a system architecture containing three elements $\mathbb{E}$, $\mathbb{E}'$, and $\mathbb{E}''$

Theorem 1 and Definition 3 sets the basis for describing the main contribution, i.e., how formal support can be provided for tasks (I)–(IV) in the context of authoring specifications. This section presents the main contribution, but prior to doing so, the considered context will first be described.

3.1 Application context: authoring specifications

This section describes the considered context for the proposed support. This context is based on the underlying idea of structuring specifications for a system in accordance with the system architecture. This system can be of any domain, e.g., SW, HW, mechanical, electrical, etc, and also a heterogeneous system, i.e., a system composed of parts from multiple domains. Structuring the specifications for a system in accordance with its architecture is an established idea [11, 79] that is also advocated in FuSa standards such as ISO 26262.

More specifically, in the considered context, specifications, containing requirements, are to be *allocated* to a specific element in the architecture; a link *Alloc*, specifying what element that a specification S is allocated to, is assumed to be contained in the specification. The allocation of a specification has the meaning that its contained requirements are allocated to the element to which the specification is allocated to. A specification, allocated to a specific element, also contain a set of variables X^{spec} that is intended to specify the interface of that element.

Requirements in different specification can also be linked with each other in accordance with Sect. 2.3.2, and the information regarding the linkage is contained in the specifications. More precisely, a specification contains two sets *Assu* and *Full* for each contained requirement R in the specification. The set *Assu* contains all incoming 'Assumption of' trace links to R and *Full* contains all outgoing 'Fulfills' trace links from R.

Figure 5 shows an example intended to illustrate the underlying idea of the context, as explained above. The example involves three specifications S, S', and S'', structured with respect to an architecture containing three elements $\mathbb{E}$, $\mathbb{E}'$, and $\mathbb{E}''$ where $\mathbb{E}'$ is the root element. Specification S, allocated to element $\mathbb{E}'$, contains a requirement R. Requirement R has an outgoing 'Fulfills' trace link to a requirement R', contained in specification S', and an incoming 'Assumption of' trace link from a requirement R'', contained in specification S''. Specification S also contains a set X^{spec}, specifying the interface of element $\mathbb{E}$.

Section 3.1.1, which now follows, will formalize what has been mentioned so far in this section.

3.1.1 Specifications

A system is specified by a set of specifications $\{S_i\}_{i=1}^{N}$ where each S_i is structured as an ordered set:

$$\left(Alloc_i, X_i^{spec}, \{(Assu_{i,j}, Full_{i,j}, \mathsf{R}_{i,j})\}_{j=1}^{N_i}\right), \text{ where}$$

- $Alloc_i$ is a variable, called *the allocation of* S_i, that is either *NIL*,[1] symbolizing that it does not have a value, or equal to an element representation;
- X_i^{spec} is a possibly empty set of variables called the *interface-specifying set of* S_i;
- $\mathsf{R}_{i,j}$ is a requirement assertion;
- each $Assu_{i,j}$ is a possibly empty set of incoming 'Assumption of' trace links from requirements in $\bigcup_{i=1}^{N}\{\mathsf{R}_{i,1}, \ldots, \mathsf{R}_{i,N_i}\}$ to $\mathsf{R}_{i,j}$; and
- each $Full_{i,j}$ is a possibly empty set of outgoing 'Fulfills' trace links from $\mathsf{R}_{i,j}$ to requirements in $\bigcup_{i=1}^{N}\{\mathsf{R}_{i,1}, \ldots, \mathsf{R}_{i,N_i}\}$.

Let $\mathcal{A}_{i,j}$ denote the set of requirements with outgoing 'Assumption of' trace links to $\mathsf{R}_{i,j}$ in accordance with $Assu_{i,j}$. Each specification S_i specifies a set of contracts $\{(\mathcal{A}_{i,1}, \mathsf{R}_{i,1}, X_i^{spec}), \ldots, (\mathcal{A}_{i,N_i}, \mathsf{R}_{i,N_i}, X_i^{spec})\}$ intended to be for the element $Alloc_i$. This also means that if the allocation of a specification is equal to an element, then this means that each requirement, contained in the specification, is allocated to this element.

Consider organizing the set of requirements $\bigcup_{i=1}^{N}\{\mathsf{R}_{i,1}, \ldots, \mathsf{R}_{i,N_i}\}$ as nodes in a graph in accordance with each set $Assu_{i,j}$ and $Full_{i,j}$. Assume, as part of the context, that the specifications in $\{S_i\}_{i=1}^{N}$ are such that this graph is a DAG; that is, it is assumed that sets $Assu_{i,j}$ and $Full_{i,j}$ do not contain arcs that result in cyclic dependencies. From this assumption, it follows that this graph is a contract structure if conditions (i)–(v) of Definition 3 holds. Notably, three of these conditions, namely conditions (i)–(iii) already hold

[1] Contraction of the Latin term *nihil*, meaning nothing.

Table 1 Formalization of tasks in context of authoring specifications

(I) Specifying 'allocation of requirements to architecture elements,' i.e., specifying each $Alloc_i$ such that:

$$Alloc_i = (X_i, \mathsf{B}) \text{ where } (X_i, \mathsf{B}) \in \mathscr{A} \tag{6}$$

(II) Specifying 'interfaces of architecture elements,' i.e., specifying each X_i^{spec} to be in accordance with condition (4) (of condition (ii) of Theorem 1):

$$X_i^{spec} = X_i \text{ if } Alloc_i = (X_i, \mathsf{B}) \tag{7}$$

(III) Specifying 'requirements,' i.e., specifying each requirement $\mathsf{R}_{i,j}$ such that:

condition (5) (of condition (ii) of Theorem 1) holds, i.e.,

$$X_{\mathsf{R}_{i,j}} \subseteq X_{Env_{\mathscr{E}}(\mathbb{E}_i)} \cup X_i \text{ if } Alloc_i = (X_i, \mathsf{B}) \text{ ; and} \tag{8}$$

condition (iii) of Theorem 1 holds, i.e.,

$$\bigcap_{k=1}^{N_k} \mathsf{R}'_k \subseteq \mathsf{R}_{i,j} \tag{9}$$

where $\{\mathsf{R}'_1, \ldots, \mathsf{R}'_{N_k}\}$ is the set of direct predecessors of $\mathsf{R}_{i,j}$ with 'Fulfills' arcs to $\mathsf{R}_{i,j}$

(IV) Specifying 'trace links between requirements,' i.e., specifying each set $Assu_{i,j}$ and $Full_{i,j}$ such that:

each assumption in each set $\mathcal{A}_{i,j}$ is in accordance with conditions (a) and (b) of Definition 3, i.e., each assumption is either:

(a) a guarantee of a contract for a sibling of $\mathbb{E}_i$; or

(b) an assumption of a contract for a proper ancestor of $\mathbb{E}_i$,

if the set of requirements $\bigcup_{i=1}^{N} \{\mathsf{R}_{i,1}, \ldots, \mathsf{R}_{i,N_i}\}$ is organized as nodes in a graph in accordance with each set $Assu_{i,j}$ and $Full_{i,j}$, then this graph is in accordance with conditions (iv) and (v) of Definition 3, i.e., the graph is such that:

(iv) if there is an arc labeled 'Fulfills' from $\mathsf{R}_{i,j}$ to $\mathsf{R}_{k,l}$, then $\mathsf{R}_{k,l}$ is a guarantee of a contract for a proper ancestor of $\mathbb{E}_i$; and

(v) if a guarantee $\mathsf{R}_{i,j}$ is reachable from an assumption A of a contract for a proper ancestor $\mathbb{E}_m$ of $\mathbb{E}_i$, then A is also an assumption of any contract $(\mathcal{A}_{k,l}, \mathsf{R}_{k,l}, X_k)$ where $\mathbb{E}_k$ is a proper ancestor of $\mathbb{E}_i$ and a descendant of $\mathbb{E}_m$ (including itself) and where $\mathsf{R}_{k,l}$ is reachable from $\mathsf{R}_{i,j}$

for such a graph. That is, the specification structure in itself enforces these conditions as long as the arcs in $Assu_{i,j}$ and $Full_{i,j}$ do not result in cyclic dependencies.

Note that there are ways, other than the one described above, in which trace links between requirements can be contained in specifications. For example, trace links could be contained in specifications separate to those containing requirements, or 'Fulfills' trace links could be contained in the specification containing the requirement to which the trace links are incoming, instead of outgoing. However, for the sake of the proposed support it does not matter; another way of containing trace links could be chosen, as long as the graph formed from the trace links is the same.

3.1.2 Formalization of tasks in considered context

Section 3.1.1 formalized a structure for authoring specifications for a system. This structure will be assumed when presenting the proposed support for tasks (I)–(IV). As previously mentioned, the overall aim of completing such tasks is to achieve property (3), i.e., completeness, in between requirement levels. This section formalizes these tasks for the specification structure presented in Sect. 3.1.1.

Given an architecture $\mathscr{A}$ and a set of specifications $\{S_i\}_{i=1}^{N}$, consider organizing the set of requirements $\bigcup_{i=1}^{N} \{\mathsf{R}_{i,1}, \ldots, \mathsf{R}_{i,N_i}\}$ as nodes in a graph in accordance with each set $Assu_{i,j}$ and $Full_{i,j}$. From the previously considered assumption that this graph is a DAG, it follows that this graph is in accordance with conditions (i)–(iii) of Definition 3.

Now consider the effort of achieving property (3), i.e., completeness, between each requirement level specified by $\{S_i\}_{i=1}^{N}$ using Theorem 1. In accordance with the specification structure described in Sect. 3.1, Definition 3, and Theorem 1, this effort consists of tasks (I)–(IV) as described in Table 1. The content of Table 1 will be described in Sect. 3.2.

3.1.3 Semiformal representations of requirements and architecture

In Sect. 1, requirements and architectures were formally defined based on the notion of an assertion, i.e., set of runs. However, in an industrial setting, requirements and architecture would typically not be represented formally, but rather informally, e.g., as free text, or semiformally, e.g., as a model with a defined syntax but without a well-defined semantics. This section describes and formalizes semiformal representations of requirements and architectures.

An architecture $\mathscr{A}$ is represented semiformally if it can only be determined that condition (a)–(i) and (b) of Definition 1 hold and not condition (a)–(ii). This means that it is possible to distinguish the hierarchical structuring of elements and their interfaces, but not their behaviors.

A requirement R *is represented semiformally* if the set of variables constrained by R can be distinguished, but not its

set of runs. For example, the requirement R_{LMsys}, shown in Fig. 2, can be represented semiformally as free text embedding distinguishable port variables l and f:

> Level l, presented by the level meter, shall be equal to actual level f in the tank.

These semiformal representations will, as part of the considered context, be assumed to be used to describe architectures and requirements in specifications.

3.2 Feedback- and guidance-driven support for tasks (I)–(IV)

Consider the context described in Sect. 3.1, or more specifically, consider a set of specifications, as described in Sect. 3.1.1, containing only semiformal representations of requirements for a system described by a semiformal representation of an architecture $\mathscr{A}$. Given this context, this section describes how formal support can be provided for each of the tasks (I)–(IV). This is achieved by enforcing contract conditions associated with these tasks; the conditions that can be evaluated in the considered context are those that are *syntactic*, i.e., those that can be evaluated despite the requirements and architecture representations being semiformal, and not formal.

This support is considered to be of two different types, *feedback* and *guidance*. *Feedback* is considered to be the cases where a user, authoring specifications, is notified that content in a specification violates one of the syntactic contract conditions. *Guidance* covers the cases where it is distinguished to the user, prior to inserting content, that certain content is in accordance with these syntactic contract conditions and some content is not.

Note that the violation of a syntactic contract condition in a specification S can, in special cases, be identified even prior to specifying *Alloc*, i.e., the allocation of specification S. However, the focus in the rest of this section will be on describing the support provided for task (II)–(IV) for authoring S whenever *Alloc* has indeed been specified to be equal to an element. Violations that can be identified whenever *Alloc* is *NIL* will in any case be identified when an element is eventually specified to be *Alloc*.

3.2.1 Task (I): Specifying allocation of requirements to architecture elements

As shown in Table 1, the formal interpretation of task (I) is a contract condition (i.e., condition 6) that is violated whenever the allocation $Alloc_i$ of a specification S_i is *NIL* or is equal to an element that is not in architecture $\mathscr{A}$. This condition can indeed be evaluated despite the fact that $\mathscr{A}$ is represented semiformally, which means that this condition is a syntactic contract condition. The fact that this condition is syntactic is shown in Table 2. This table also shows which of the contract conditions of task (II)–(IV) that are syntactic; the table will be fully justified in subsequent sections.

Table 2 Identification of the contract conditions in Table 1 to either be syntactic or not

Task	Formal condition(s)	Syntactic?
(I)	Condition (6) in Table 1	Yes
(II)	Condition (7) in Table 1	Yes
(III)	Condition (8) in Table 1	Yes
	Condition (9) in Table 1	No
(IV)	Conditions (a) and (b) and (iv)–(v) in Table 1	Yes

This section now proceeds to describe the support that can be provided by evaluating condition 6 in Table 1.

Let $Alloc_i$ be specified to be an element (X, B) in architecture $\mathscr{A}$. Notably, considering the contract conditions that can be evaluated in Table 1, i.e., those that are syntactic in accordance with Table 2, it is possible that certain conditions can be evaluated *directly* after specifying the allocation, but not prior. This means that, whenever $Alloc_i$ is *NIL*, it is possible to guide a user in specifying the allocation $Alloc_i$ by distinguishing to the user between which elements (in the architecture) that $Alloc_i$ can and cannot be equal to without violating the syntactic contract conditions associated with tasks (II)–(IV). Consider the following example.

Example 1 Consider a set of specifications containing requirements and trace links specified in accordance with the graph shown in Fig. 2 except that the requirement R_{lMeter} has not yet been allocated. That is, the requirement R_{lMeter} is in a specification S_i where $Alloc_i$ is *NIL*. Prior to specifying $Alloc_i$, it is possible to distinguish, to a user authoring S_i, that if $Alloc_i$ is equal to, e.g., $\mathbb{E}_{Esys}$ instead of $\mathbb{E}_{lMeter}$, then conditions (a)–(b) in Table 1 will be violated since $\mathbb{E}_{Esys}$ is not in the environment of $\mathbb{E}_{lMeter}$ in this architecture. □

Example 1 showed how guidance can be provided when specifying the allocation of a specification by distinguishing to a user that conditions (a)–(b) in Table 1 would be violated if the allocation were to be equal to a specific element. However, more generally, such guidance can be provided not only by evaluating conditions (a)–(b), but rather all of the contract conditions that are in Table 1 and are syntactic in accordance with Table 2.

3.2.2 Task (II): Specifying architecture element interfaces consisting of port variables

As shown in Table 1, condition (7) is the only contract conditions associated with task (II) and this condition is a

syntactic contract condition in accordance with Table 2. This condition is syntactic since whenever the allocation $Alloc_i$ of a specification S_i is equal to an element (X_i, B_i) in architecture $\mathscr{A}$, it can be evaluated whether or not the interface X_i is equal to the interface-specifying set X_i^{spec} in S_i without knowing the behaviors of the elements in $\mathscr{A}$. As will be shown in this section, support when authoring S_i can be provided by evaluating this condition. An example now follows.

Example 2 Let S_{LMsys} be a specification containing an interface-specifying set $X_{LMsys}^{spec} = \{f\}$ and that the allocation of S_{LMsys} is equal to the element $\mathbb{E}_{LMsys}$, shown in Fig. 1b. Through evaluation of condition (7) in Table 1, it possible to give feedback, to a user authoring S_{LMsys}, that the interface $X_{LMsys} = \{f, l\}$ of $\mathbb{E}_{LMsys}$ and the interface-specifying set X_{LMsys}^{spec} are not equal; it is also possible to provide guidance to the user by distinguishing which port variables that are not in the interface-specifying set, but only in the interface, and vice versa. □

As shown in Example 2, guidance and feedback for task (II) can be given by evaluating condition (7) in Table 1. Note that such support can be provided without acknowledging neither the interface-specifying set of a specification nor the interface, of the element to which the allocation of the specification is equal to in the architecture, to be the source of correctness. Under development, it is also only natural that they are not equal and typically there is neither need nor desire to resolve this immediately. However, at point of deployment, the interface-specifying set and the interface in the non-behavioral architecture data should be in accordance with condition (7).

3.2.3 Task (III): Specifying requirements

In accordance with Table 1, condition (8) and condition (9) of Theorem 1 are the contract conditions associated with task (III), i.e., specifying requirements. While condition (8) can be evaluated by only knowing which port variables that the requirements constrain, to evaluate condition (9), the runs of the requirements must also be known. That is, as shown in Table 2, condition (9) is not a syntactic condition and cannot be evaluated in the given context. However, as also shown in Table 2, condition (8) is indeed syntactic and the following will describe how feedback and guidance can be provided by enforcing this condition.

Consider that the allocation of a specification is equal to an element in the architecture. By evaluating condition (8) in Table 1, it is possible to give feedback by identifying each distinguishable port variable in requirements contained in the specification where such a port variable violates condition (8). An example of when condition (8) is violated is shown in Fig. 4.

Furthermore, when specifying requirements in a specification where its allocation is equal to an element in the architecture, it is possible to guide a user by distinguishing between port variables that a requirement can and cannot be specified over in order for condition (8) in Table 1 to hold. An example of this now follows.

Example 3 Consider a user specifying a requirement in a specification where its allocation is equal to the element $\mathbb{E}_{Esys}$ in the architecture shown in Fig. 1b. It is possible to give guidance to the user by distinguishing that specifying the requirement over, e.g., the set $\{f, v_{branch}\}$ will violate condition (8) in Table 1, but over $\{f, l, h\}$, it will not. □

3.2.4 Task (IV): Specifying trace links between requirements

As expressed in Table 1, conditions (a), (b), (iv)–(v) are the conditions associated with task (IV). As previously mentioned in Sect. 3.1.2, given that circular dependencies are not specified between requirements, conditions (i)–(iii) of Definition 3 automatically hold for the given context. Regarding conditions (a), (b), (iv)–(v), all of these can be evaluated without knowing the runs of neither requirements nor elements, and thus, as shown in Table 2, these conditions are syntactic. In fact, these conditions can actually be evaluated *regardless of the representation format of the requirements*. Thus, by considering these conditions when specifying 'Assumption of' and 'Fulfills' trace links to and from requirements in a specification S_i, i.e., specifying the sets $Assu_i$ and $Full_i$, it is possible to guide the user by distinguishing between trace links that will and will not violate conditions (a), (b), (iv)–(v) in Table 1. Additionally, feedback can be provided if already established trace links violate these conditions. Examples of when these conditions are violated are presented in Sect. 4.6 in the context of an industrial example system.

3.2.5 Applicability of proposed support

Sections 3.2.1–3.2.1 described the feedback- and guidance-driven support that can be provided for tasks (I)–(IV) by enforcing the syntactic contract conditions in Table 2. As previously mentioned, this support was described given the context in Sect. 3.1. This section discusses the applicability of the considered context and the proposed support.

Regarding the context, it can be noted that it does not impose any constraints neither on the order in which specifications are authored nor on the order in which content is entered within a specification. That is, a specification for a leaf element in the specification can be authored prior to a specification for

the root element, and vice versa. Also, the requirements in a specification can be specified prior to specifying the interface-specifying set, and the other way around. This means that the context, and thus also the proposed support, is indeed not restricted to any type of design flow: top-down, bottom-up, or anything in between. As previously mentioned in Sect. 1, such flexibility caters to the fact that different developers have dissimilar, yet successful, approaches for optimizing systems development; a top-down design flow might be optimal when, e.g., development is outsourced, but may be less optimal in other cases.

Regarding the assumption concerning the use of semi-formal representations, while much of the proposed support is indeed available even when informal representations of requirements are used, semiformal or formal representations of both requirements and architecture are required to achieve full support. However, if working toward achieving stringent RE, then a higher level of stringency in representations might be needed anyway, and in safety standards such as ISO 26262 and DO-178C for the avionic domain, semiformal or formal representations are actually required. All in all, the context is in essence nothing more than a formalized setting for achieving stringent RE, which means that the proposed support is, at least in concept, applicable for any company working with stringent RE, driven, e.g., by FuSa.

3.3 Additional condition-enforcing support

This section discusses how to provide additional support by enforcing conditions, other than the syntactic conditions in Table 2, from the theory described in Sect. 2. Notably, out of the conditions in Table 2, condition (9) of Theorem 1 was the only condition that was not discussed in Sect. 3.2 since this condition cannot be evaluated unless requirements are represented formally. However, if requirements are indeed represented formally, e.g., in the language Temporal Logic of Actions $(TLA)^+$ [45], then an approach such as [81] can be used to evaluate condition (9) of Theorem 1.

As previously mentioned in Sect. 2.4, the work in [75] presents formal definitions of SIL assignment given a contract structure for an architecture. These definitions can be found in Appendix. In the same manner that the syntactic contracts are evaluated to provide support in Sect. 3.2, these definitions can also be evaluated to provide support for assigning SILs to requirements; notably, these definitions can also be evaluated regardless of the representation format of requirements and behaviors of elements in the architecture. An example will be presented in Sect. 4.6.

4 Implementation of specification tool in an industrial setting

As previously mentioned in Sect. 1, as an initial form of validation, an industrial case study was performed at trucks manufacturer Scania where the support presented in Sect. 3.2 was implemented in a specification tool, which was also integrated into Scania's development tool chain. This section will describe and draw conclusions from this case study, but first, the overall goal and challenges faced when implementing this specification tool are presented.

4.1 Goal and challenges of industrial case study

The main goal of the case study was to evaluate the potential of realizing the support presented in Sect. 3.2 in an industrial setting. Since the proposed support assumes the context described in Sect. 3.1, realizing the support is dependent on that both the tool support and the context are implemented. Formulated as queries, the main challenges faced when implementing the context were:

(A) Do architecture data in a *semiformal* format exist? (Do machine-readable architecture representations exist where data on the hierarchical structuring of elements and their interfaces can be extracted? If such representations do not exist, can these data be obtained in some other way?)
(B) If architecture data in a semiformal format exist, potentially in different formats and stored in different databases and tools, how can the data be extracted and combined into an *overall* semiformal architecture representation?
(C) How to enable and manage links from specifications to architecture data and between requirements in the same or different specifications?

Similarly, the main challenge when implementing the proposed tool support was:

(D) How to evaluate syntactic contract conditions over specification and architecture data, possibly distributed over different databases and tools?

Addressing queries (A)–(D) evaluates whether or not it is feasible for the proposed support to be *technically* implemented in an industrial setting and is considered to provide an initial form of validation of the proposed support.

The validation would be initial since even if a solution technically addresses the challenges formulated by queries (A)–(D), the solution may still suffer from having

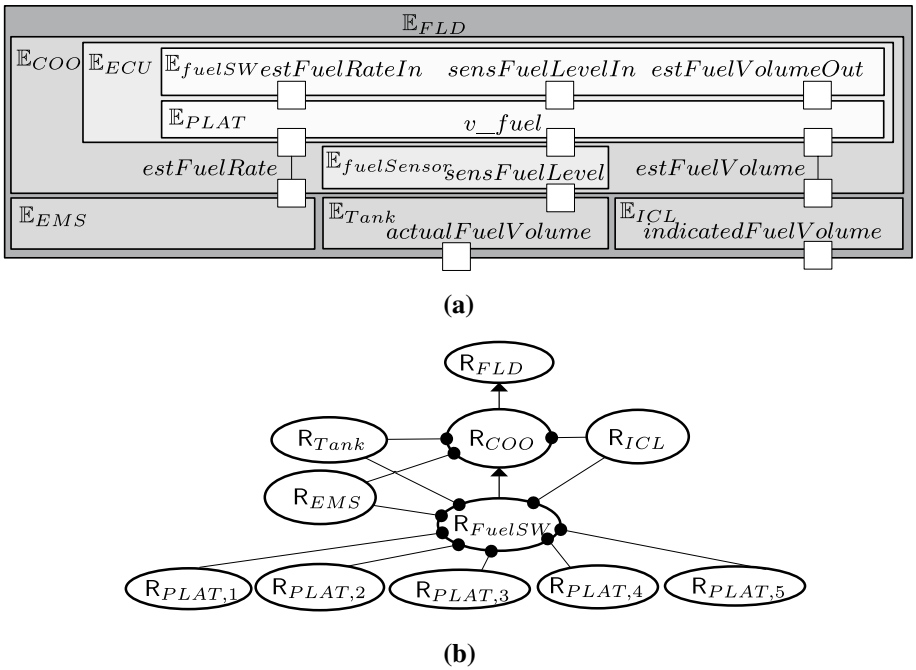

Fig. 6 An architecture of FLD and safety requirements in a contract structure for this architecture, respectively

a low degree of usability, which means that such a solution would still not be viable in an industrial setting. For example, if the number of actions required to embed a reference to a port variable in a requirement (to specify it semiformally) is too high, then engineers will most likely not specify requirements semiformally, and will thus not get the full benefits from the proposed support. With this in mind, usability was considered as a critical aspect when implementing the specification tool. However, evaluating usability requires long-term empirical studies involving engineers working toward achieving stringent RE, and given that stringent RE is not currently practiced to a large extent, such studies are currently difficult to perform. Therefore, the focus of the present paper is on addressing queries (A)–(D), rather than on evaluating the aspect of usability. However, despite not being adequately evaluated, the present paper will still report on and discuss how the aspect of usability was taken into account in the implementation of the specification tool.

Now that the overall goal and challenges of the industrial case study have been presented, the present paper will proceed in describing the case study. This case study also included working with an industrial example system, which will first be introduced. This industrial example will in the following be used instead of the simplistic LM-system that has been used for illustrating theoretical concepts. After describing the case study, an evaluation section follows where queries (A)–(D) are addressed.

4.2 Fuel level display system

Fuel level display (FLD) is a safety-critical system installed on all trucks manufactured by Scania, with a functionality to provide an estimate of the fuel volume in the fuel tank to the driver. The system is safety-critical since running out of fuel results in loss of power steering which, in turn, makes a heavy truck near impossible to steer. FLD will be described in terms of an architecture and a contracts structure for this architecture as shown in Fig. 6a, b, respectively.

4.2.1 FLD architecture

As shown in Fig. 6a, FLD $\mathbb{E}_{FLD}$ consists of a fuel tank $\mathbb{E}_{Tank}$ and three electric control unit (ECU)-systems, i.e., an ECU with sensors and actuators: Engine Management System (EMS) $\mathbb{E}_{EMS}$; Instrument Cluster (ICL) $\mathbb{E}_{ICL}$; and Coordinator (COO) $\mathbb{E}_{COO}$. In turn, $\mathbb{E}_{COO}$ is composed of a fuel sensor $\mathbb{E}_{fuelSensor}$ and an ECU $\mathbb{E}_{ECU}$, which consists of an application SW component $\mathbb{E}_{FuelSW}$ and a platform $\mathbb{E}_{PLAT}$, i.e., ECU HW and infrastructure SW, which $\mathbb{E}_{FuelSW}$ executes on. Due to space restrictions, only a breakdown of one ECU-system is considered and this breakdown is also limited; see [74] for a more complete architecture.

The element $\mathbb{E}_{COO}$ estimates the fuel volume *actualFuelVolume* in the tank $\mathbb{E}_{Tank}$ by a Kalman filter that is implemented by $\mathbb{E}_{fuelSW}$. The platform $\mathbb{E}_{PLAT}$ is to ensure that the inputs *estFuelRateIn* and *sensFuelLevelIn* and output

estFuelVolumeOut to $\mathbb{E}_{FuelSW}$ correspond to the inputs *estFuelRate* and *sensFuelLevel* and output *estFuelVolume* of $\mathbb{E}_{COO}$, respectively. The port variable *sensFuelLevel* represents the position of a floater in the fuel tank $\mathbb{E}_{FuelTank}$, as sensed by the fuel sensor $\mathbb{E}_{fuelSensor}$ and *estFuelRate* is an estimate of the current rate of fuel injected into the engine and is a controller area network (CAN) signal, transmitted in CAN message *FuelEconomy* from $\mathbb{E}_{EMS}$. The estimated fuel volume is transmitted as the CAN signal *estFuelVolume* in CAN message *DashDisplay*. This CAN message is received by $\mathbb{E}_{ICL}$ where a fuel gauge *indicatedFuelVolume* in the display presents the information to the driver.

4.2.2 Contract structure for FLD architecture

Each requirement shown in Fig. 6b is a safety requirement where the subscript of each safety requirement denotes which element the requirement is *allocated* to, e.g., R_{FLD} is allocated to $\mathbb{E}_{FLD}$. These safety requirements are, instead of being represented formally as in the examples of assertions in Sect. 2.1, represented *semiformally* as free text with formal references to port variables. As an example of a requirement, the overall safety requirement R_{FLD} on FLD is represented semiformally as:

> *indicatedFuelVolume*, shown by the fuel gauge, is less than or equal to *actualFuelVolume*.

As another example, the safety requirement R_{ICL}, which is allocated to $\mathbb{E}_{ICL}$, is represented semiformally as

> *indicatedFuelVolume* corresponds to *estFuelVolume*.

4.3 Referencing and dereferencing using linked data

Now that the FLD system has been described, the design and implementation of the specification tool follow.

Each specification authored in the specification tool is structured to contain the data described in Sect. 3.1.1, i.e., the allocation of the specification, its interface-specifying set, requirements, and their trace links. In addition to data contained in a specification, data from another specification or even from another tool can be presented when opening the specification in the specification tool. This is achieved through *referencing* this other data in accordance with Linked Data [9]: inserting a reference means inserting a Uniform Resource Locator (URL) that is associated with the data that are to be presented; the URL does not only contain the data that are to be presented, but rather information on how the data should be retrieved. Inserted URLs can be *dereferenced* in a standardized manner using Hypertext Transfer Protocol (HTTP).

Whenever a URL is inserted, the specification tool will first retrieve the data and then present it accordingly. Note that dereferencing may be done in several steps since URLs associated with certain data may also contain URLs associated with other data. In accordance with the specification structure, each data object contained in a specification, e.g., requirements, their trace links, port variables, is associated with a URL. Dereferencing of URLs is done automatically by the specification tool, thus, ensuring that the presented data are *consistent*, i.e., updated with the data that are associated with the URL. Consider the following example.

Example 4 Assume that architecture data in accordance with Fig. 6a are made available as Linked Data. Consider inserting an URL associated with the port variable *indicatedFuelVolume* [%] with the intent of specifying the requirement R_{FLD} semiformally. Upon insertion, the URL is dereferenced, which results in that the name of the port variable is presented in the specification and made to be distinguishable as a port variable. If the data associated with this address is modified, e.g., if *indicatedFuelVolume* [%] changes name to *indFuelLevel*[%], then upon refreshing the specification containing the requirement R_{FLD}, this change will be immediately reflected in the specification. □

As shown in Example 4, Linked Data enable data in specifications to be consistent. This is a crucial property in an industrial setting. Consider that data in specifications are manually updated or imported/exported between specifications rather than being linked in accordance with Linked Data. In practice, ensuring that all referenced data between them are continuously updated is unmanageable even for a relatively small set of specifications.

4.4 Using RDF for publishing and consuming linked data

The following terminology will be used throughout the rest of this paper: a tool is said to *publish* data if this tool makes data available to other tools in accordance with Linked Data; and if a first tool dereferences addresses associated with data published by a second tool, then the first tool is said to *consume* data from the second tool.

In practice, in order to publish and consume data, a standardized underlying data model is needed and this model is typically Resource Description Framework (RDF) [43]. Accordingly, the specification tool publishes the data contained in specifications as RDF triples, consisting of a subject (URL), a predicate (URL), and an object (URL or a literal). Consider that these published data are represented as a graph with subjects and objects as nodes and predicates as arcs, which will in the following be referred to as *links*. The

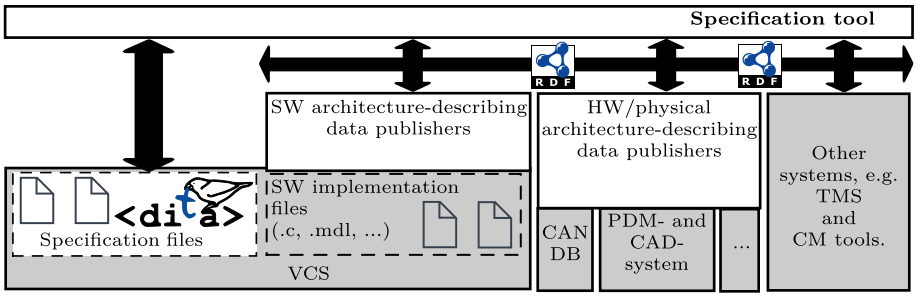

Fig. 7 Integration of specification tool into the industrial tool chain at Scania

specification tool publishes data such that, for each specification, it holds that:

- for each contained requirement R in the specification:
  - each 'Assumption of' trace link, from a requirement R′ to R, corresponds to a link from R′ to R;
  - each 'Fulfills' trace link, from R to another requirement R′, corresponds to a link from R to R′; and
  - each distinguishable port variable in R corresponds to a link from R to a port variable in architecture data;
- each member in the interface-specifying set where this member is a reference to a port variable in architecture data corresponds to a link from the interface-specifying set to this port variable in architecture data; and
- the allocation of the specification, whenever containing a reference to an element in architecture data, corresponds to a link from each requirement in the specification to this element.

Using Linked Data and RDF also enables the language SPARQL[2] [60] for querying data. The specification tool uses SPARQL for evaluating the syntactic contract conditions in Table 2. More specifically, these evaluations are done by traversing and dereferencing links between data, and thus, the evaluations are over consistent data.

Note that so far, the source of the architecture data has not been described. This will be done in Sect. 4.5, which follows after this section. However, the fact is that, when using the Linked Data approach, the source of the architecture data is not important; the source could be the specification tool itself, an external tool, or several external tools, whichever are used for modeling the system architecture.

[2] A recursive acronym for SPARQL Protocol and RDF Query Language.

4.5 Integration of specification tool into industrial tool chain

The integration of the specification tool into the tool chain at Scania is shown in Fig. 7 where arrows represent flow of data and where tools/aspects preexisting the integration and new tools/aspects of the tool chain are color-coded with gray and white, respectively.

Regarding the data flow shown furthest to the left in Fig. 7, this bidirectional arrow represents the saving and loading of specifications; each specification is saved as/loaded from exactly one file, called a *specification file*. Specification files are saved and loaded as Extensible Markup Language (XML)-files in accordance with Darwin Information Typing Architecture (DITA) [30]. DITA is an open standard for authoring specifications and publishing them as, e.g., PDF-documents.

Specification files are stored in the preexisting version control system (VCS) along with *SW implementation files*, i.e., source code files (e.g., .c-files) and files (e.g., Simulink [17] .mdl files) that generate source code. Relying on preexisting VCS allows versions of specifications and SW to automatically coevolve since new versions of specifications are automatically created whenever SW development is branched/merged.

The arrows other than the one furthest to the left in Fig. 7 capture publishing/consumption of data as described in Sect. 4.4. In general, any tool can consume/publish data; for example, test management systems (TMS) can consume data published by the specification tool and link additional information to the requirements such as who, where, when, and how requirements have been or will be tested. As another example, the specification tool can consume data from change management (CM) tools and link requirements to change requests. More importantly, the specification tool consumes architecture data published by other tools in the tool chain.

Consider published architecture data. In the following, the present paper will distinguish between such data that either: (i) *describes* the *implemented* system; or (ii) *specifies* the *intended* architecture of a system. In the following, data of

class (i) and (ii) will be referred to as *architecture-describing data* and *architecture-specifying data*, respectively.

As examples of such distinction between data, data extracted from a .h-file associated with a .c-file could be architecture-describing data and architecture-specifying data could be data in a high-level architecture model, represented in a language such as SysML, Unified Modeling Language (UML) [67], or Architecture Analysis and Design Language (AADL) [24]. However, this architecture model could also describe an implemented system; thus, this categorization into architecture-describing and architecture-specifying data is *subjective*. The important aspect is that this categorization is made clear in a specific tool chain. The following will describe how this categorization was done in the industrial case study.

Regarding publishing of SW architecture-describing data, these data are obtained from tools, i.e., publishers, which automatically analyze and extract data directly from SW implementation files in VCS using *architecture recovery* [62, 82]. For example, SW variables and the functions that read and write to them are extracted from parsing .h and .c-code files. Relying on architecture recovery ensures that the published data are *consistent* with SW files. In addition, published architecture-describing data are also linked to version data.

In a similar manner, HW/physical architecture-describing data publishers automatically analyze and extract data from production-based sources containing data that describe the implemented parts of the system that is not SW. Examples of these sources are the product data management (PDM) system, which lists the elements present in a particular vehicle; databases, e.g., CAN-DB, which lists CAN messages and signals; and other sources, e.g., Excel-files describing properties of sensors and actuators and computer-aided design (CAD)-systems.

4.6 Authoring support in specification tool

With respect to authoring specifications, the specification tool functions similar to a typical text editor, e.g., Microsoft Word, where sections and tables, and also images and equations can be removed or embedded in free text simply by inserting them from the menu. As shown in Fig. 7, the visual design of the user interface (UI) of the specification tool is also similar to a typical text editor, only the requirements appear differently and are distinguishable as rectangles filled with light gray as shown in Fig. 7.

The rest of this section will describe the support provided by the specification tool when authoring specifications; this includes both the feedback- and guidance-driven support for tasks (I)–(IV) described in Sect. 3.2, as well as other type of support that increases quality of specifications. However, prior to presenting this support, a principle is introduced for how the syntactic contract conditions in Table 2 are evaluated.

This principle is that the syntactic contract conditions are evaluated considering architecture-describing data rather than architecture-specifying data. This principle is motivated by the fact that at *the point of deployment of a system*, requirements and other specification data should express intended properties of the *implemented* system and not properties of models that describe the system as it was intended to be implemented.

However, *during development of a system*, it might be the case that certain architecture-describing data are not available; in an early design phase in particular, it might be the case that only architecture-specifying data are available. Thus, during development, it might be desirable or even necessary to, e.g., express requirements over port variables in the architecture-specifying data or to manually specify interfaces instead of inserting references to architecture-describing data. This is indeed allowed by the specification tool; in fact, the specification tool will generally not restrict the user from entering data into a specification; the specification tool will simply identify if syntactic contract conditions are violated with respect to the architecture-describing data.

In summary, a user is free to refer to or enter data other than architecture-describing data in, e.g., requirements—as needed during system development. However, these data should *eventually* be replaced with architecture-describing data references and the specification tool will therefore continuously alert the user until this is done.

4.6.1 Task (I): Specifying allocation of requirements to architecture elements

As explained in Sect. 3.1, specifying the allocation of a specification to an element means allocating the requirements in the specification to this element. Specifying the allocation is incorporated into the specification tool by having an option where the user is prompted to select, from a list of architecture elements, the architecture element that the allocation of the specification is to be equal to. In accordance with Sect. 3.2.1, when selecting from this list, the specification tool will guide the user by distinguishing between elements (in the architecture data) that the allocation can and cannot be equal to without violating the syntactic contract conditions associated with tasks (II)–(IV).

Specifying the allocation of a specification can be done at any time after creating the specification. Prior to specifying the allocation, the tool will continuously urge the user to specify the allocation.

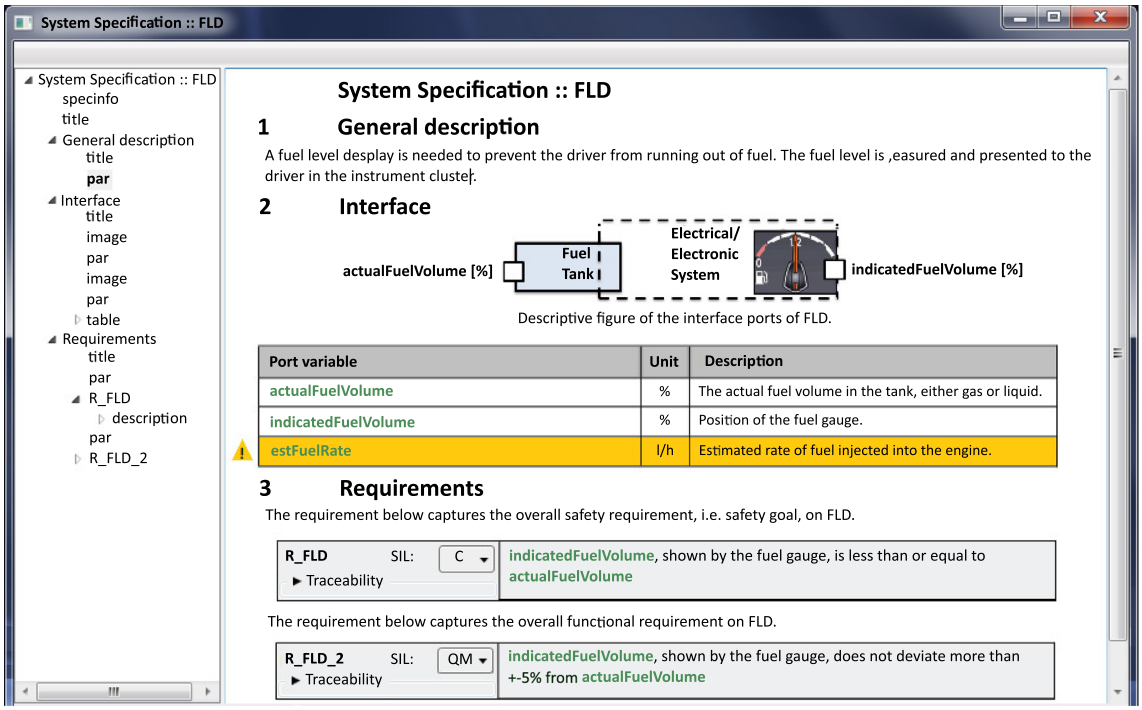

Fig. 8 Snapshot (tweaked for enhanced readability) of the main window of the UI of the implemented specification tool

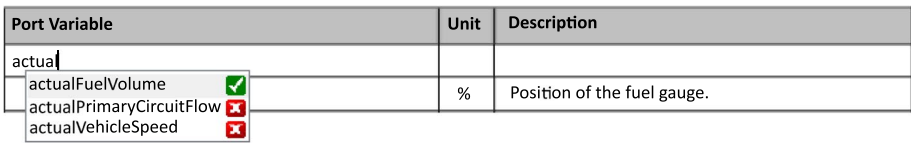

Fig. 9 Entering data into interface table using auto-complete functionality

4.6.2 Task (II): Specifying architecture element interfaces consisting of port variables

Specifying the interface-specifying set of a specification is realized by a concept called *interface table*, which specifies interface port variables and their properties. In general, a specification can have several interface tables, typically one for each port variable type, e.g., CAN signals, sensor inputs, etc., since their properties and thus the number of desired columns in the table may differ. The union of the interface tables is the interface-specifying set. An example of an interface table is shown in Fig. 8.

When entering data into interface tables, the user can input references to port variables in the architecture-describing data using *auto-complete functionality*. An example of the auto-complete functionality is shown in Fig. 9 when entering data into the table also shown in Fig. 8. Upon entering a reference, the specification tool will automatically dereference and present, not only the name of the port variable as given in the architecture-describing data, but also other relevant properties.

Whenever the allocation of the specification is equal to an element in the architecture-describing data, in accordance with Sect. 3.2.2, the specification tool will evaluate condition (7) of Table 1 and give feedback and guidance. An example is shown in Fig. 8 where the user is provided with the feedback (table row marked yellow and warning triangle) that the port variable *estFuelRate* is not a port variable of $\mathbb{E}_{FLD}$, as described by the consumed architecture-describing data that are in accordance with Fig. 6a.

Notably, as shown in Fig. 9, the auto-complete functionality also guides the user by distinguishing between which of the port variables that are and are not in accordance with condition (7) of Table 1. For example, in Fig. 9, the port variables that are and are not in accordance with condition (7) are followed by a check-mark and crosses, respectively. Additionally, there is an option to automatically populate entire interface tables with references to the interface port variables in architecture-describing data in accordance with condition (7), thus ensuring consistency and saving much manual and error-prone work.

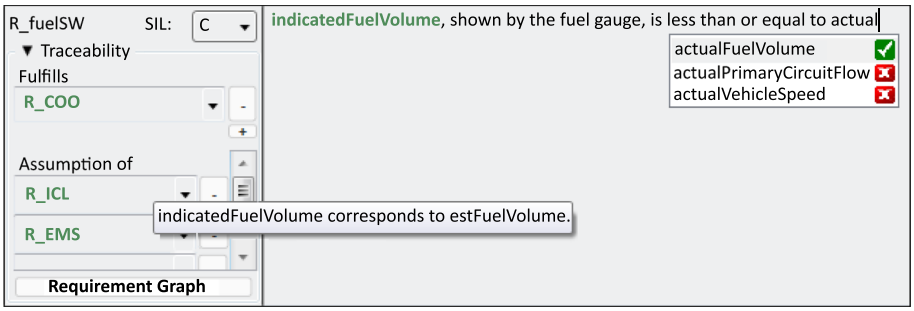

Fig. 10 Snapshot (tweaked for enhanced readability) of a requirement in the implemented specification tool

4.6.3 Task (III): Specifying requirements

In the specification tool, as previously indicated, there is an option to insert requirements into a specification. The user is free to specify a requirement as seen fit; even images and equations can be embedded.

Explicit support is given for specifying requirements in semiformal representation as free text with references to port variables; two examples of such a representation format were presented in Sect. 4.2.2. This support for specifying requirements in semiformal representation is given by having auto-complete functionality for entering port variable references. Figure 10 shows an example where a list of port variable names appear as possible references when specifying the safety requirement R_{fuelSW}.

Similar to the support provided for task (II) and in accordance with Sect. 3.2.3, whenever the allocation of a specification is equal to an element, the specification tool will give feedback on requirements specified prior to their allocation. That is, the tool will flag port variable references that violate condition (8) of Table 1. Furthermore, as shown in Fig. 10, the auto-complete functionality will guide the user by distinguishing to the user between port variables (listed by the auto-complete functionality in consumed architecture-describing data) that are and not are in accordance with condition (8).

4.6.4 Task (IV): Specifying trace links between requirements

In the specification tool, for a requirement R in a specification, there are options to specify 'Assumption of' and 'Fulfills' trace links between R and other requirements. As shown in Fig. 10, this is done by selecting sources for 'Assumption of' trace links and targets for 'Fulfills' trace links for R from a fold-down menu with lists for specifying such trace links; hovering over a specified trace link source or target and the requirement reference is dereferenced and its representation is presented, as exemplified in Fig. 10 for the requirement R_{ICL}. As also shown in Fig. 10, there is an option to view published data on requirements and their links as a navigable 'Requirements Graph' (similar to Fig. 6b) that shows R in the context of neighboring requirements that are traceable from or to R through specified requirement trace links.

In accordance with Sect. 3.2.4, whenever the allocation of a specification is equal to an architecture element, the specification tool will guide the user in specifying trace links between R and other requirements by distinguishing between 'Assumption of' trace link sources and 'Fulfills' trace link targets that are and are not in accordance with conditions (a), (b), (iv), and (v) in Table 1. As previously mentioned in Sect. 3.2.4 and as will be exemplified below, each of these conditions can be evaluated regardless of the representation format of requirements.

As a first example of how the specification tool can provide guidance, consider a specification that contains R_{COO} and where the allocation of the specification is equal to $\mathbb{E}_{COO}$. Consider also that all the requirements in the graph shown in Fig. 6b are allocated as described in Sect. 4.2.2. In accordance with conditions (a) and (b), when selecting 'Assumption of' trace link sources for the requirement R_{COO}, only the requirements R_{tank}, R_{ICL}, and R_{EMS} from this graph will appear as sources distinguishable as being in accordance with conditions (a) and (b). The other requirements in the graph, e.g., $\mathsf{R}_{PLAT,1}$, will be distinguished to violate these conditions since these requirements are not allocated to an element in the environment of $\mathbb{E}_{COO}$, to which R_{COO} is allocated.

As a second example, consider specifying 'Fulfills' trace links for the requirement R_{fuelSW} shown in Fig. 10. Given that each requirement $\mathsf{R}_{PLAT,i}$ has been allocated to $\mathbb{E}_{PLAT}$, each of these requirements will be distinguished to violate condition (iv) in Table 1. Condition (iv) is violated since $\mathbb{E}_{PLAT}$ is not a proper ancestor of $\mathbb{E}_{fuelSW}$, to which R_{fuelSW} is allocated.

Similar to condition (v) in Table 1, enforcing condition (v) also allows distinguishing between the selection of possible 'Fulfills' trace link targets. An example of a case where adding a 'Fulfills' trace link would violate condition (v) was previously presented in Sect. 2.3.2 and is shown in Fig. 3.

Note that the specification tool will allow any trace links to be specified as long as these do not result in cyclic dependencies; even when trace links result in that syntactic contract conditions are violated. However, whenever

specified trace links violate these conditions, the tool will provide feedback by notifying the user and flagging these trace links.

4.6.5 Condition-enforcing support for specifying SILs

As shown in Fig. 10, SILs can be specified for requirements in a specification. The specification tool evaluates the definitions of SIL assignment in Appendix to be able to give feedback on SILs specified for requirements. Without getting into specifics, an example of where these definitions would be violated is if the requirement R_{COO} shown in the contract structure in Fig. 6b is assigned with a lower SIL than the requirement R_{FLD}.

4.7 Evaluation of tool support

This section describes how the implementations solutions used in the industrial case study addressed queries (A)–(D), which can, as previously mentioned in Sect. 4.1, be considered as criteria for evaluating technical implementability of the support described in Sect. 3.2. This section also discusses these solutions from a usability perspective.

Regarding query (A) 'Do architecture data in a semiformal format exist?', it would have been convenient if semiformal architecture representations, together providing an *accurate* description of the entire truck, would have been *directly* available, e.g., as SysMl models. However, in an industrial setting such as the development tool chain of Scania, architecture representations are predominantly informal, not consistent with system implementations, and containing errors. However, while not being available directly, semiformal architecture-describing data were found to be indirectly available in implementation artifacts, e.g., code, and in production-based sources such as the PDM-system in the Scania case.

Consider query (B): 'If architecture data in a semiformal format exist, potentially in different formats and stored in different databases and tools, how can the data be extracted and combined into an overall *semiformal architecture representation?'*. Since it cannot be assumed that semiformal architecture-describing data are not available directly in an industrial setting, these data would have to first be made available. As described in Sect. 4.5, this was achieved by the use of adapters that publish architecture data retrieved through *architecture recovery* [62, 82], i.e., through parsing and analyzing implementation artifacts and production-based sources. Considering that an individual adapter only would typically publish only a part of the architecture of a system, for the data published by the different adapters to form an overall architecture, the data need to be linked together. This can be achieved by relying on a common data model, e.g., an RDF schema, which enables such linkage given that each publisher is compliant with the subset of the data model that concerns its published data.

Regarding queries (C) 'How to enable and manage links from specifications to architecture data and between requirements in the same or different specifications?' and (D) 'How to evaluate syntactic contract conditions over specification and architecture data, possibly distributed over different databases and tools?', as described in Sects. 4.3 and 4.4, the proposed solution, which is considered to be general, was to use Linked Data. As previously mentioned in Sect. 4.4, evaluation of the syntactic contract conditions can then be performed by traversing and dereferencing links, and thus, the evaluation can be performed over consistent data.

Regarding performing evaluations over consistent data, this requires that each reference is dereferenced every time an evaluation is made. Since traversal can sometimes by extensive and the number of dereferences high for the evaluation of a single contract condition, making sure that the evaluation is over consistent data was observed to sometimes come at the cost of slower performance of the specification tool. Fast performance is however a critical factor for ensuring the usability of the feedback- and guidance-driven support described in Sect. 4.6. That is, it is considered critical that a user gets immediate feedback when violating a contract condition such that it is clear what action caused the violation, and for guidance, that the choices are distinguishable almost directly to avoid slowing down the user. To enhance performance, the specification tool was therefore implemented such that it sometimes caches dereferenced data locally and re-uses these data when doing evaluations. Due to this re-use of old data, there is a slight possibility that the specification tool exhibits unsound behavior, e.g., providing a warning when there is no violation of a contract condition. Thus, a general observation is that there seems to be a trade-off between performance and performing evaluations over the most consistent data—investigations are currently ongoing for trying to find the right balance.

Usability was, as previously mentioned in Sect. 4.1, considered as a critical factor when implementing the specification tool. One aspect of usability has already been discussed above, namely performance. Other usability aspects concern features implemented in the specification tool. One such feature is auto-complete functionality, which supports specifying semiformal requirements with little additional effort than specifying informal requirements. Another usability-related feature is the hovering functionality, which allows a user to quickly get a description of referenced data without tracing the reference back to its source.

In summary, the case study indicated that the proposed support is indeed possible to technically implement, even when semiformal architecture data are not available directly. Thus, despite the need for further validation, especially long-term evaluation of usability, the case study showed high

potential in realizing the proposed support in an industrial setting.

5 Related work

As mentioned in Sect. 1, there already exist approaches where contract theory is used to provide tool support. For example, contracts theory is used to provide tool support for safety analyzes in [19], safety certification in [69], model-based design in [5], and failure propagation modeling in [56, 57]. More similar to the present paper, the work [13, 14] presents tool support for verifying contracts *refinement*, which in essence corresponds to verifying completeness. Even more similar to the present paper, the work [16] describes tool support for verifying a hierarchical organization of contracts related to a system architecture model. Despite the similarities between [13, 14, 16] and the present paper, as previously mentioned in Sect. 1, while the tool support in [13, 14, 16] requires that contracts must be formally represented in the language linear-time temporal logic (LTL) [59], the tool support described in the present paper does not require that contracts are specified in formal representation; in fact, the present paper describes how explicit support can be provided when requirements are specified in semiformal representation.

There are also other works [23, 27, 44] that focus on providing feedback- and/or guidance-driven tool support for specification, albeit with a fundamentally different approach from the present paper. That is, in contrast to the present paper where support is provided by enforcing formal conditions, the works in [23, 27, 44] rely on natural language (NL) processing to provide feedback and guidance on requirements represented in NL considering, e.g., text length and terms usage with respect to a domain ontology/dictionary. Hence, while the approach in [23, 27, 44] improves readability of requirements, the approach in the present paper enforces their correctness, and thus, the approaches complement each other. NL processing is also used in [50, 51] to automatically cluster and create trace links between requirements. One main difference between support relying on NL processing and the support proposed in the present paper is that while the former cannot in general be guaranteed to be sound, the support in the present paper only enforces necessary conditions and will, thus, never generate false-positives.

In contrast to [23, 27, 44, 50, 51], but in accordance with the present paper, the works [2, 3, 28, 29] describe formally founded support for RE. Similar to the present paper, the works [3, 28] both focus on establishing trace links between requirements and design/architecture. However, while the approach in the present paper is applicable for any type design flow, i.e., top-down, bottom-up, or anything in between, the approach in [3] requires using model transformations for driving development and is therefore only applicable for a top-down design flow. In contrast to the present paper, but similar to the previously mentioned contract-based approaches [13, 14, 16], the support in [3, 28] relies on formal representations of architectures and requirements. The work in [2] describes a formal model-based development methodology using requirements refinement. However, analogous to [3], this methodology is only applicable to a top-down design flow and also prescribes the use of formal models and requirements. In comparison, the support in the present paper is applicable in a greater context and this is to cater to needs in current state of industrial practice where requirements and architectures are typically not represented formally and where design flows vary, sometimes even within the same company.

Regarding the relation between the present paper and the work in [29], while both providing support for authoring specifications, the present paper and [29] have different focuses and complement each other; the present paper focuses on tool support for tasks (I)–(IV) while [29] focuses on transformation between requirements specified in NL, the formal representation format Object Constraint Language (OCL) [71], and a semiformal representation format in between these.

There also exist tool support [20, 65] for Goal-Oriented Requirements Engineering (GORE), see, e.g., I* [80] or Knowledge Acquisition in Automated Specification (KAOS) [46] or [34, 47] for literature reviews, where [46, 80] draw on ideas presented in [39, 40, 58]. While such tool support is indeed similar to the support proposed in the present paper, one main difference lies in the flexibility of how requirements can be hierarchically structured. More specifically, in GORE models, the use of assumptions, also called *expectations*, cannot be used at lower requirement levels and are strictly limited to constrain the environment of a *SW* system. This is in contrast to the present paper where the proposed support builds on a contract structure, which can be used to structure requirements for a system in any domain and where assumptions can be used at all requirement levels.

6 Conclusion

This paper has presented tool support applicable when working with stringent RE, e.g., as advocated by FuSa standards such as IEC 61508 and ISO 26262. Despite the need for further evaluations, especially long-term empirical studies, an industrial case study showed high potential in realizing the proposed support in an industrial setting.

More specifically, as the main contribution, by evaluating syntactic conditions from established contracts theory, it has been shown how feedback- and guidance-driven support can be given for tasks (I)–(IV) when authoring specifications in

accordance with a system architecture. It has been shown that such support can be given for structuring requirements, regardless of their representation format. In particular, if requirements are expressed with references to port variables in an architecture describing the implemented system, then feedback- and guidance-driven can be provided for both specifying and structuring requirements. Furthermore, with the use of the proposed guided auto-complete functionality for input of references to port variables in architecture-describing data, transforming requirements specified in NL to incorporate such references is straightforward. Notably, this approach caters to needs in current state of industrial practice where requirements and architectures are typically not represented formally. Furthermore, moving to specifying requirements containing such references also allows powerful analyzes over data on requirements and architecture to answer queries such as 'what requirements are enforced on my CAN signal or SW-variable?'.

A proposed concept for enabling such analyzes in practice is Linked Data, which supports formal referencing and dereferencing of data in between specifications and architecture data. In accordance with the Linked Data approach, it has been shown that input of references to architecture-describing data, not just in requirements, but also in, e.g., interface tables, allows to enforce specifications to be consistent with these data. Moreover, if architecture-describing data are obtained through architecture recovery, as in the industrial case study, this actually means enforcing specifications to be consistent with the SW implementation. Notably, such consistency is not only a mandatory property when attempting to achieve FuSa, but rather a highly desirable property of specifications in general.

Hence, not only has this paper described how to provide tool support for the stringent RE effort advocated by FuSa standards, but also how to increase quality of specifications in general.

Funding Funding was provided by VINNOVA (Grant No. 2011-04446).

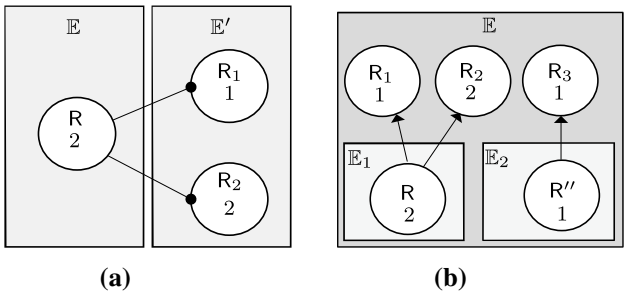

Fig. 11 Two representative examples of SIL inheritance are shown in context of contract structures

Appendix: Extending contracts theory with safety integrity levels

This section presents an extension of contracts theory, as described in Sect. 2, with the notion of SILs. The contained material can be considered as a compressed and polished version of the work in [75].

Definition of SIL

In Sects. 1 and 2.4, a SIL was described as a measure of the required reliability of a system or component, in order to achieve a tolerable level of *risk*, i.e., the combination of the probability of harm and the severity of such a harm. Formulated differently in the context of specifying a SIL for a requirement, the SIL specifies the tolerable level of risk of *violating* the requirement. Thus, prior to presenting a formal definition of a SIL in the context of contracts theory, it is necessary to define what it means for a requirement to be violated.

Given an architecture $\{(X_i, \mathsf{B}_i)\}_{i=1}^{N}$ and a contract $(\mathcal{A}, \mathsf{R}, X)$ for an element in the architecture, *the guarantee* R *is violated* if a run ω of $\bigcap_{i=1}^{N} \mathsf{B}_i$ is executed where $\{\omega\} \nsubseteq \mathsf{R}$.

Definition 4 (*Safety integrity level for guarantee*) Given an architecture $\mathscr{A}$ and a contract $(\mathcal{A}, \mathsf{R}, X)$ for an element in $\mathscr{A}$, a *SIL for the guarantee* R, denoted SIL_{R}, is a uniquely specified discrete level that corresponds to a target range of the probability that the guarantee R is violated, during an arbitrary time interval of a predefined length. □

Definition 4 is in accordance with the definitions in ISO 26262 and IEC 61508, given that a 'safety function' in IEC 61508 is, or can at least correspond to, a top-level requirement and that an automotive SIL (ASIL) for a requirement in ISO 26262 can be mapped to a range of the probability that the requirement is violated. SILs range from 1 to 4 in IEC 61508 and ASILs from *A*–*D* in ISO 26262.

Consider that a SIL is assigned to the requirement R_{LMsys}, as shown in Fig 2, as a result of assessing the risk of $\mathbb{E}_{LMsys}$ in the context of a specific architecture. Following a certain standard, this would imply that the specific instructions of that standard would have to be followed with the aim of achieving a failure probability of R_{LMsys} within the target range as specified by the SIL for R_{LMsys}.

SIL inheritance in contract structure

As described in Sect. 1, in ISO 26262, safety requirements are to be structured at different hierarchical levels with the intent that safety requirements at one level are to fulfill all safety requirements at the higher level. Consider that two lower-level safety requirements are to fulfill a higher-level safety requirement. If the higher-level safety requirement has been specified with a SIL *D*, for example, then the two lower-level safety requirements will *inherit* SIL *D*. If one of the lower-level safety requirements is also derived from another higher-level safety requirement, then that the lower-level safety requirement will inherit the highest SIL of the higher-level safety requirement.

In this section, the concept of SIL inheritance will be formally defined in terms of a contract structure. Prior to presenting such a formal definition, the concept is introduced informally by the use of the following representative examples (a) and (b), as shown in Fig. 11.

(a) In Fig. 11a, the intent is that the behavior of $\mathbb{E}'$ is to fulfill R_1 and R_2, given that the behavior of $\mathbb{E}$ fulfills R. Therefore, the highest SIL for R_1 and R_2, i.e., 2, is specified for R.
(b) In Fig. 11b, the intent is that the requirement R, allocated to the child $\mathbb{E}$ of an element $\mathbb{E}'$, is to fulfill both requirements R_1 and R_2, allocated to $\mathbb{E}$. Hence, the highest SIL for R_1 and R_2, i.e., 2, is specified for R. The requirement R'' allocated to $\mathbb{E}''$ is intended to fulfill only R_3 and is thus specified with the SIL 1.

As expressed in examples (a) and (b), if the intent is to rely on that a requirement R is not violated in order to ensure that a requirement R_i is not violated, then the requirement R should *inherit* the SIL of R_i. For example, the behavior of $\mathbb{E}_{Esys}$ cannot fulfill the requirement R_{Esys} unless the potentiometer is installed correctly in the tank, as expressed by R_{tank} and shown in Fig. 2. Thus, the SIL for R_{tank} is inherited from R_{Esys}.

A formal definition of SIL inheritance will now follow. Note that to fully understand the formal definition, SIL inheritance needs to be explained simultaneously with SIL decomposition. Hence, a detailed explanation will follow in 'Appendix: SIL decomposition in contract structure' section.

Definition 5 (*SIL inheritance in contract structure*) Given an architecture $\mathscr{A}$ and a set of contracts $\mathfrak{C}$ organized as contract structure for $\mathscr{A}$, *SIL inheritance* is the specification of a SIL to a guarantee R in a contract $C \in \mathfrak{C}$ for an element $\mathbb{E} \in \mathscr{A}$ such that $SIL_{\mathsf{R}} = max(SIL_{\mathsf{R}_1}, \ldots, SIL_{\mathsf{R}_N})$ where each R_i is either:

(a) a guarantee without any incoming arcs labeled 'Fulfills' and R is a direct predecessor of R_i and an assumption in a contract where R_i is the guarantee; or
(b) a guarantee in a contract for an ancestor of $\mathbb{E}$ and a direct successor of R. □

Part (a) and (b) of Definition 5 corresponds to examples (a) and (b), respectively.

SIL decomposition in contract structure

Consider a case where either one of two safety requirements can, in fact, fulfill another higher-level safety requirement alone and that the safety requirements are allocated to two elements that are 'sufficiently independent,' i.e.,

> absence of failures whose probability of simultaneous or successive occurrence cannot be expressed as the simple product of the unconditional probabilities of each of them, between two or more elements that could lead to the violation of a safety requirement, or organizational separation of the parties performing an action [37].

Given such a case, it is possible to assign lower SILs to the safety requirements than to the higher-level safety requirement by applying 'SIL decomposition,' i.e.,

> apportioning of safety requirements redundantly to sufficiently independent elements, with the objective of reducing the ASIL of the redundant safety requirements that are allocated to the corresponding elements [37].

In this section, the concept of SIL decomposition in the context of a contract structure is presented. As previously described, SIL decomposition can only be performed if redundancy is present in a system. In order to capture the intent of achieving redundancy, the definition of a contract structure, i.e., Definition 3, is extended with an '$OR_{\perp}$' node. Two representative examples follow immediately after the definition to explain the extension.

Definition 6 (*Extension of contract structure for architecture*) Given an architecture $\mathscr{A}$ and a set $\bigcup_{i=1}^{N}\{(\mathcal{A}_{i,1}, \mathsf{R}_{i,1}, X_i), \ldots, (\mathcal{A}_{i,N_i}, \mathsf{R}_{i,N_i}, X_i)\}$ where each

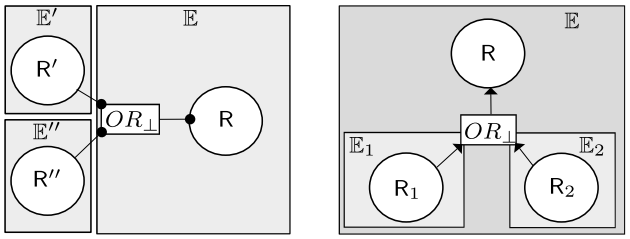

Fig. 12 Two representative examples are shown where the intent is to achieve redundancy

$(\mathcal{A}_{i,j}, \mathsf{R}_{i,j}, X_i)$ is a contract for an element $\mathbb{E}_i$ of $\mathscr{A}$ and where each assumption in each set $\mathcal{A}_{i,j}$ is either:

(a) a guarantee of a contract for a sibling of $\mathbb{E}_i$; or
(b) an assumption of a contract for a proper ancestor of $\mathbb{E}_i$,

then a *contract structure* for $\mathscr{A}$ is an arc-labeled DAG, such that:

(i) each node is either a guarantee $\mathsf{R}_{i,j}$ or an '$OR_\perp$' node and each guarantee $\mathsf{R}_{i,j}$ is a node;
(ii) each arc is uniquely labeled either 'Assumption of" or "Fulfills';
(iii) if and only if $\mathsf{R}_{k,l}$ is in $\mathcal{A}_{i,j}$, then there exists an arc labeled 'Assumption of' from a node $\mathsf{R}_{k,l}$ to either:
  (a) $\mathsf{R}_{i,j}$; or
  (b) an '$OR_\perp$' node that has exactly one outgoing arc to $\mathsf{R}_{i,j}$, labeled 'assumption of';
(iv) if there is an arc labeled 'Fulfills' from $\mathsf{R}_{i,j}$ to $\mathsf{R}_{k,l}$, then $\mathsf{R}_{k,l}$ is a guarantee of a contract for a proper ancestor of $\mathbb{E}_i$; and
(v) if a guarantee $\mathsf{R}_{i,j}$ is reachable from an assumption A of a contract for a proper ancestor $\mathbb{E}_m$ of $\mathbb{E}_i$, then A is also an assumption of any contract $(\mathcal{A}_{k,l}, \mathsf{R}_{k,l}, X_k)$ where $\mathbb{E}_k$ is a proper ancestor of $\mathbb{E}_i$ and a descendant of $\mathbb{E}_m$ (including itself) and where $\mathsf{R}_{k,l}$ is reachable from $\mathsf{R}_{i,j}$;
(vi) if an '$OR_\perp$' node has an incoming arc labeled 'Fulfills' from $\mathsf{R}_{i,j}$, then the '$OR_\perp$' node has exactly one outgoing arc to a guarantee $\mathsf{R}_{k,l}$ of a contract for a proper ancestor of $\mathbb{E}_i$; and
(vii) each '$OR_\perp$' node has at least two incoming arcs and where any two incoming arcs to the '$OR_\perp$' node, are guarantees of contracts for two different elements.

□

The two following representative examples (a) and (b), also shown in Fig. 12a, b, capture two scenarios with the intent to achieve redundancy.

(a) As expressed in Fig. 12a, the intent is that it is sufficient that either one of the behaviors of two 'sufficiently independent' [37] elements $\mathbb{E}'$ and $\mathbb{E}''$ in the environment to an element $\mathbb{E}$ fulfills the respective requirements R' and R'', in order for the behavior of $\mathbb{E}$ to fulfill R, i.e., that $(\mathsf{R}' \cup \mathsf{R}'') \cap \mathsf{B} \subseteq \mathsf{R}$.
(b) As expressed in Fig. 12b, the intent is that either one of two requirements R_1 and R_2 on two sufficiently independent children $\mathbb{E}_1$ and $\mathbb{E}_2$ of an element $\mathbb{E}$ is able to fulfill a requirement R of a contract for $\mathbb{E}$, i.e., that $(\mathsf{R}_1 \cup \mathsf{R}_2) \subseteq \mathsf{R}$.

Considering examples (a) and (b), it is hence possible to assign a potentially lower SIL to R' and R'', and also to R_1 and R_2, than the SIL for R, by performing SIL decomposition.

Definition 7 (*SIL decomposition in contract structure*) Given an architecture $\mathscr{A}$ and a set of contracts $\mathfrak{C}$ organized as contract structure for $\mathscr{A}$, *SIL Decomposition* is the specification of a SIL to a guarantee R in a contract $C \in \mathfrak{C}$ for an element $\mathbb{E} \in \mathscr{A}$ such that $\beta \leq SIL_{\mathsf{R}} \leq max(SIL_{\mathsf{R}_1}, \ldots, SIL_{\mathsf{R}_N})$, where each R_i is either:

(a) a guarantee without any incoming arcs labeled 'Fulfills' and a direct successor of an $OR_\perp$ node where $OR_\perp$ node has an incoming 'Assumption of' arc from R; or
(b) a direct successor of an $OR_\perp$ where $OR_\perp$ node has an incoming 'Fulfills' arc from R,

and where β is determined by the context and in accordance with a given safety standard. □

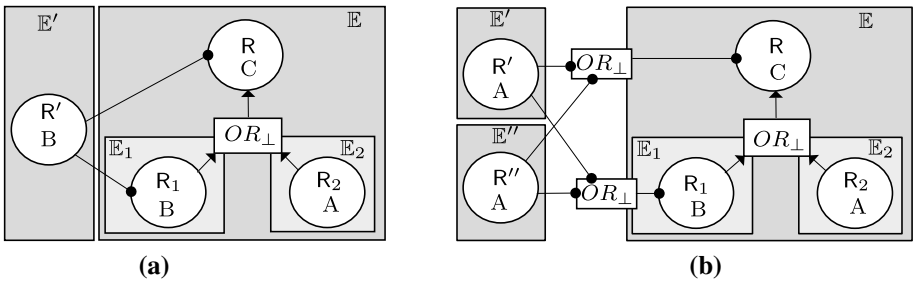

Fig. 13 Examples are shown where both SIL inheritance and decomposition are applied

Consider Fig. 13a that is intended to clarify why inheritance of SILs, through the use of assumptions, only apply to guarantees without any incoming 'Fulfills' arcs, as expressed in part (a) of Definition 5. If SIL C has been assigned to R, for example, then SIL B and A can be assigned to R_1 and R_2, respectively, according to part (b) of Definition 7 and ISO 26262. According to part (a) of Definition 5, R' should inherit SIL B from the requirement R_1 without any incoming 'Fulfills' arcs, rather than the SIL for R, since redundancy has been introduced into $\mathbb{E}$. The same reasoning can be applied to decomposition of SILs, through the use of assumptions, as shown in Fig. 13b and expressed in part (a) of Definition 7.

Remark 2 Although not explicitly mentioned in either ISO 26262 nor IEC 61508, if both SIL decomposition and inheritance can be applied to a requirement R, then the maximum SIL should be assigned to R. □

References

1. Abadi M, Lamport L (1993) Composing specifications. ACM Trans Program Lang Syst 15(1):73–132. https://doi.org/10.1145/151646.151649
2. Abrial JR, Butler M, Hallerstede S, Hoang TS, Mehta F, Voisin L (2010) Rodin: an open toolset for modelling and reasoning in Event-B. Int J Softw Tools Technol Transf 12(6):447–466
3. Almeida JPA, Iacob ME, Van Eck P (2007) Requirements traceability in model-driven development: applying model and transformation conformance. Inf Syst Frontiers 9(4):327–342
4. Bauer S, David A, Hennicker R, Guldstrand Larsen K, Legay A, Nyman U, Wąsowski A (2012) Moving from specifications to contracts in component-based design. In: Lara J, Zisman A (eds) Fundamental approaches to software engineering. Lecture notes in computer science, vol 7212. Springer, Berlin, pp 43–58. https://doi.org/10.1007/978-3-642-28872-2_3
5. Baumgart A, Reinkemeier P, Rettberg A, Stierand I, Thaden E, Weber R (2011) A model-based design methodology with contracts to enhance the development process of safety-critical systems. In: Min SL, Pettit R, Ungerer T (eds) Software technologies for embedded and ubiquitous systems. Lecture notes in computer science, vol 6399. Springer, Berlin, pp 59–70. https://doi.org/10.1007/978-3-642-16256-5_8
6. Benveniste A, Caillaud B, Ferrari A, Mangeruca L, Passerone R, Sofronis C (2008) Multiple viewpoint contract-based specification and design. In: de Boer F, Bonsangue M, Graf S, de Roever WP (eds) Formal methods for components and objects. Lecture notes in computer science, vol 5382. Springer, Berlin, pp 200–225. https://doi.org/10.1007/978-3-540-92188-2_9
7. Benveniste A, Caillaud B, Nickovic D, Passerone R, Raclet JB, Reinkemeier P, Sangiovanni-Vincentelli A, Damm W, Henzinger T, Larsen KG (2012) Contracts for system design. Rapport de recherche RR-8147, INRIA. http://hal.inria.fr/hal-00757488
8. Benveniste A, Caillaud B, Passerone R (2009) Multi-viewpoint state machines for rich component models. In: Nicolescu G, Mosterman P (eds) Model-based design for embedded systems. Taylor & Francis, pp 487–518 http://www.google.se/books?id=8Cjg2mM-m1MC
9. Bizer C, Heath T, Berners-Lee T (2009) Linked data-the story so far. In: Semantic services, interoperability and web applications: emerging concepts, pp 205–227
10. Böschen M, Bogusch R, Fraga A, Rudat C (2016) Bridging the gap between natural language requirements and formal specifications. In: Joint proceedings of REFSQ–2016 workshops, doctoral symposium, research method track, and poster track (REFSQ–JP 2016), CEUR workshop proceedings, pp 1–11. CEUR–WS. http://ceur--ws.org/Vol--1564/paper20.pdf
11. Broy M (2017) A logical approach to systems engineering artifacts: semantic relationships and dependencies beyond traceability-from requirements to functional and architectural views. Softw Syst Model. https://doi.org/10.1007/s10270-017-0619-4
12. Cheng BHC, Atlee JM (2007) Research directions in requirements engineering. In: Future of software engineering, 2007. FOSE '07, pp 285–303. https://doi.org/10.1109/FOSE.2007.17
13. Cimatti A, Dorigatti M, Tonetta S (2013) Ocra: A tool for checking the refinement of temporal contracts. In: 2013 IEEE/ACM 28th international conference on automated software engineering (ASE), pp 702–705. https://doi.org/10.1109/ASE.2013.6693137
14. Cimatti A, Tonetta S (2015) Contracts-refinement proof system for component-based embedded systems. Sci Comput Program 97(Part 3):333–348. https://doi.org/10.1016/j.scico.2014.06.011
15. Cleland-Huang J, Gotel O, Zisman A (2012) Software and systems traceability. Springer, Berlin
16. Cofer D, Gacek A, Miller S, Whalen MW, LaValley B, Sha L (2012) Compositional verification of architectural models. In: Proceedings of the 4th international conference on NASA formal methods, NFM'12. Springer, Berlin, pp 126–140. https://doi.org/10.1007/978-3-642-28891-3_13
17. Dabney JB, Harman TL (2004) Mastering simulink. Pearson/Prentice Hall, Upper Saddle River
18. Damm W (2005) Controlling speculative design processes using rich component models. In: Fifth international conference on application of concurrency to system design, 2005. ACSD 2005, pp 118–119. https://doi.org/10.1109/ACSD.2005.35
19. Damm W, Josko B, Peinkamp T (2009) Contract based ISO CD 26262 safety analysis. In: Safety-critical systems, 2009. SAE. https://doi.org/10.4271/2009-01-0754
20. Darimont R, Delor E, Massonet P, van Lamsweerde A (1997) GRAIL/KAOS: an environment for goal-driven requirements engineering. In: Proceedings of the (19th) international conference on software engineering, pp 612–613. https://doi.org/10.1145/253228.253499
21. Diestel R (2012) Graph theory. Graduate texts in mathematics, vol 173, 4th edn. Springer, Berlin
22. EN 50128: Railway applications—communication, signalling and processing systems—software for railway control and protection systems (2011)
23. Farfeleder S, Moser T, Krall A, Stålhane T, Zojer H, Panis C (2011) DODT: increasing requirements formalism using domain ontologies for improved embedded systems development. In: 2011 IEEE 14th international symposium on design and diagnostics of electronic circuits systems (DDECS), pp 271–274. https://doi.org/10.1109/DDECS.2011.5783092
24. Feiler PH, Gluch DP (2012) Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language, 1st edn. Addison-Wesley Professional, Boston
25. Firesmith D (2004) Engineering safety requirements, safety constraints, and safety-critical requirements. J Object Technol 3(3):27–42
26. Friedenthal S, Moore A, Steiner R (2008) A practical guide to SysML: systems modeling language. Morgan Kaufmann Inc., San Francisco
27. Génova G, Fuentes JM, Llorens J, Hurtado O, Moreno V (2013) A framework to measure and improve the quality of textual

requirements. Requir Eng 18(1):25–41. https://doi.org/10.1007/s00766-011-0134-z
28. Goknil A, Kurtev I, Van Den Berg K (2014) Generation and validation of traces between requirements and architecture based on formal trace semantics. J Syst Softw 88(C):112–137. https://doi.org/10.1016/j.jss.2013.10.006
29. Hähnle R, Johannisson K, Ranta A (2002) An authoring tool for informal and formal requirements specifications. In: Proceedings of the 5th international conference on fundamental approaches to software engineering, FASE '02, pp 233–248. Springer, London. http://dl.acm.org/citation.cfm?id=645370.651289
30. Harrison N (2005) The Darwin information typing architecture (DITA): applications for globalization. In: Proceedings of the international professional communication conference, 2005, IPCC 2005. IEEE, pp 115–121
31. Henzinger T, Sifakis J (2007) The discipline of embedded systems design. Computer 40(10):32–40. https://doi.org/10.1109/MC.2007.364
32. Hoare CAR (1969) An axiomatic basis for computer programming. Commun ACM 12(10):576–580. https://doi.org/10.1145/363235.363259
33. Hooman J, de Roever WP (1986) The quest goes on: a survey of proofsystems for partial correctness of CSP. In: de Bakker JW, de Roever W-P, Rozenberg G (eds) Current trends in concurrency, overviews and tutorials. Springer, Berlin, pp 343–395. https://doi.org/10.1007/BFb0027044
34. Horkoff J, Aydemir FB, Cardoso E, Li T, Maté A, Paja E, Salnitri M, Piras L, Mylopoulos J, Giorgini P (2017) Goal-oriented requirements engineering: an extended systematic mapping study. Requir Eng. https://doi.org/10.1007/s00766-017-0280-z
35. Hull MEC, Jackson K, Dick J (eds) (2011) Requirements engineering, 3rd edn. Springer, Berlin
36. International Electrotechnical Commission: IEC 61508—functional safety of electrical/electronic/programmable electronic safety-related systems (2010)
37. International Organization for Standardization: ISO 26262—"Road vehicles-Functional safety" (2011)
38. International Organization for Standardization, International Electrotechnical Commission, Institute of Electrical and Electronics Engineers: ISO/IEC/IEEE 42010—system and software engineering—Architecture description (2011)
39. Jackson M (1995) Software requirements & specifications: a lexicon of practice, principles and prejudices. ACM Press/Addison-Wesley Publishing Co., New York
40. Jackson M (1995) The world and the machine. In: Proceedings of the 17th international conference on software engineering, ICSE '95. ACM, New York, pp 283–292. https://doi.org/10.1145/225014.225041
41. Jones CB (1983) Specification and design of (parallel) programs. In: Mason REA (ed) Information processing 83, IFIP congress series, vol 9. IFIP, North-Holland, Paris, pp 321–332
42. Josko B, Ma Q, Metzner A (2008) Designing embedded systems using heterogeneous rich components. In: Proceedings of the INCOSE international symposium
43. Klyne G, Carroll JJ (2014) Resource description framework (RDF): concepts and abstract syntax. W3C. http://www.w3.org/TR/rdf-concepts/
44. Knauss E, Lubke D, Meyer S (2009) Feedback-driven requirements engineering: the heuristic requirements assistant. In: 2009 IEEE 31st international conference on software engineering, pp 587–590. https://doi.org/10.1109/ICSE.2009.5070562
45. Lamport L (2002) Specifying systems: the TLA+ language and tools for hardware and software engineers. Addison-Wesley Longman Publishing Co., Inc., Chicago
46. van Lamsweerde A, Letier E (2004) From object orientation to goal orientation: a paradigm shift for requirements engineering. In: Wirsing M, Knapp A, Balsamo S (eds) Radical innovations of software and systems engineering in the future. Lecture notes in computer science, vol 2941. Springer, Berlin, pp 325–340. https://doi.org/10.1007/978-3-540-24626-8_23
47. Lapouchnian A (2005) Goal-oriented requirements engineering: an overview of the current research. University of Toronto, Toronto
48. Lee E (2008) Cyber physical systems: design challenges. In: 11th IEEE international symposium on object oriented real-time distributed computing (ISORC), pp 363–369. https://doi.org/10.1109/ISORC.2008.25
49. Leveson NG (1995) Safeware: system safety and computers. ACM, New York
50. Mahmoud A, Niu N, Xu S (2012) A semantic relatedness approach for traceability link recovery. In: 2012 20th IEEE international conference on program comprehension (ICPC), pp 183–192. https://doi.org/10.1109/ICPC.2012.6240487
51. Mahmoud A, Williams G (2016) Detecting, classifying, and tracing non-functional software requirements. Requir Eng 21(3):357–381. https://doi.org/10.1007/s00766-016-0252-8
52. Mcmillan KL (1999) Circular compositional reasoning about liveness. In: Advances in hardware design and verification: IFIP WG10.5 international conference on correct hardware design and verification methods (CHARME '99), volume 1703 of Lecture notes in computer science. Springer, pp 342–345
53. Meyer B (1992) Applying "design by contract". Computer 25(10):40–51. https://doi.org/10.1109/2.161279
54. Ministry of Defence: Def Stan 00-56—safety management requirements for defence systems (2007)
55. Misra J, Chandy K (1981) Proofs of networks of processes. IEEE Trans Softw Eng SE–7(4):417–426. https://doi.org/10.1109/TSE.1981.230844
56. Nyberg M (2013) Failure propagation modeling for safety analysis using causal Bayesian networks. In: 2013 conference on control and fault-tolerant systems (SysTol), pp 91–97. https://doi.org/10.1109/SysTol.2013.6693936
57. Nyberg M, Westman J (2015) Failure propagation modeling based on contracts theory. In: 2015 Eleventh European dependable computing conference (EDCC), pp 108–119. https://doi.org/10.1109/EDCC.2015.21
58. Parnas DL (1995) Functional documents for computer systems. Sci Comput Program 25:41–61
59. Pnueli A (1977) The temporal logic of programs. In: 18th annual symposium on foundations of computer science, 1977, pp 46–57. https://doi.org/10.1109/SFCS.1977.32
60. Quilitz B, Leser U (2008) Querying distributed RDF data sources with SPARQL. In: European semantic web conference. Springer, pp 524–538
61. Quinton S, Graf S (2008) Contract-based verification of hierarchical systems of components. In: Sixth IEEE international conference on software engineering and formal methods, 2008. SEFM '08, pp 377 –381. https://doi.org/10.1109/SEFM.2008.28
62. Rasool G, Asif N (2007) Software architecture recovery. Int J Comput Elec Automation Control Inf Eng 1(4):939–944
63. Rausand M, Høyland A (2004) System reliability theory: models, statistical methods, and applications. Wiley series in probability and statistics—applied probability and statistics section. Wiley, Hoboken. https://books.google.se/books?id=gkUWz9AA-QEC
64. Rawat DB, Rodrigues JJ, Stojmenovic I (2015) Cyber-physical systems: from theory to practice. CRC Press, Boca Raton
65. Rifaut A, Massonet P, Molderez JF, Ponsard C, Stadnik P, van Lamsweerde A, Hung TV (2003) FAUST: formal analysis using specification tools. In: Proceedings of the 11th IEEE international requirements engineering conference, 2003, p 350. https://doi.org/10.1109/ICRE.2003.1232781
66. de Roever W, Langmaack H, Pnueli A (1998) Compositionality: the significant difference. Springer, Berlin

67. Rumbaugh J, Jacobson I, Booch G (2004) Unified modeling language reference manual, the (2nd edition). Pearson Higher Education, London
68. Sangiovanni-Vincentelli AL, Damm W, Passerone R (2012) Taming Dr. Frankenstein: contract-based design for cyber-physical systems. Eur J Control 18(3):217–238
69. Soderberg A, Vedder B (2012) Composable safety-critical systems based on pre-certified software components. In: 2012 IEEE 23rd international symposium on software reliability engineering workshops (ISSREW), pp 343–348. https://doi.org/10.1109/ISSREW.2012.83
70. Sutcliffe A, Maiden N (1998) The domain theory for requirements engineering. IEEE Trans Softw Eng 24(3):174–196. https://doi.org/10.1109/32.667878
71. Warmer J, Kleppe A (1999) The object constraint language: precise modeling with UML. Addison-Wesley Longman Publishing Co., Inc, Boston
72. Westman J, Nyberg M (2013) A reference example on the specification of safety requirements using ISO 26262. In: Roy M (ed) Proceedings of workshop DECS (ERCIM/EWICS workshop on dependable embedded and cyber-physical Systems) of the 32nd international conference on computer safety, reliability and security, p NA. France. http://hal.archives-ouvertes.fr/hal-00848610
73. Westman J, Nyberg M (2014) Environment-centric contracts for design of cyber-physical systems. In: Dingel J, Schulte W, Ramos I, Abrahao S, Insfran E (eds) Model-driven engineering languages and systems. Lecture notes in computer science, vol 8767. Springer, Berlin, pp 218–234. https://doi.org/10.1007/978-3-319-11653-2_14
74. Westman J, Nyberg M (2015) Contracts for specifying and structuring requirements on cyber-physical systems. In: Rawat DB, Rodriques J, Stojmenovic I (eds) Cyber physical systems: from theory to practice. Taylor & Francis, Boca Raton
75. Westman J, Nyberg M (2015) Extending contract theory with safety integrity levels. In: 2015 IEEE 16th international symposium on HASE, pp 85–92. https://doi.org/10.1109/HASE.2015.21
76. Westman J, Nyberg M (2017) Conditions of contracts for separating responsibilities in heterogeneous systems. Form Methods Syst Des. https://doi.org/10.1007/s10703-017-0294-7
77. Westman J, Nyberg M, Gustavsson J, Gurov D (2017) Formal architecture modeling of sequential non-recursive C programs. Sci Comput Program 146(Supplement C):2–27. https://doi.org/10.1016/j.scico.2017.03.007
78. Westman J, Nyberg M, Törngren M (2013) Structuring safety requirements in ISO 26262 using contract theory. In: Proceedings of the 32nd international conference on computer safety, reliability, and security—volume 8153, SAFECOMP 2013, pp 166–177. Springer, New York. https://doi.org/10.1007/978-3-642-40793-2_16
79. Whalen MW, Gacek A, Cofer D, Murugesan A, Heimdahl MP, Rayadurgam S (2013) Your what is my how: iteration and hierarchy in system design. IEEE Softw 30(2):54–60. https://doi.org/10.1109/MS.2012.173
80. Yu E (1997) Towards modelling and reasoning support for early-phase requirements engineering. In: Proceedings of the third IEEE international symposium on requirements engineering, 1997, pp 226–235. https://doi.org/10.1109/ISRE.1997.566873
81. Yu Y, Manolios P, Lamport L (1999) Model checking TLA+ specifications. In: Pierre L, Kropf T (eds) Correct hardware design and verification methods: 10th IFIP WG10.5 advanced research working conference, CHARME'99 BadHerrenalb, Germany, September 27–29, 1999 proceedings, Springer, Berlin, pp 54–66. https://doi.org/10.1007/3-540-48153-2_6
82. Zhang X, Persson M, Nyberg M, Mokhtari B, Einarson A, Linder H, Westman J, Chen D, Törngren M (2014) Experience on applying software architecture recovery to automotive embedded systems. In: 2014 software evolution week-IEEE conference on software maintenance, reengineering and reverse engineering (CSMR-WCRE). IEEE, pp 379–382